VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

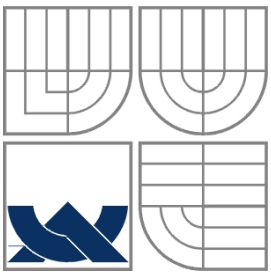
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

JEDNODUCHÁ HRA V OPENGL

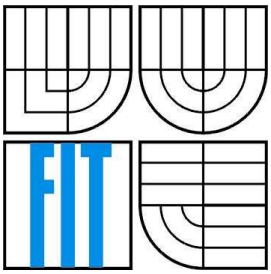
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE RADOVAN ČAPEK
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

JEDNODUCHÁ HRA V OPENGL

SIMPLE GAME USING OPENGL

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — RADOVAN ČAPEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. MIROSLAV ŠVUB
SUPERVISOR

BRNO 2008

Abstrakt

Práce se zaměřuje na problematiku počítačové grafiky a tvorby her. Popisuje fyzikální model chování auta a implementuje ho v podobě závodní hry. Hra načítá modely formátu 3ds a obsahuje mnoho zajímavých vizuálních technik. Teoretické základy těchto technik jsou v práci rozebrány. Výstupem práce je kromě programu samotného i tutoriál, jak dané techniky naprogramovat.

Klíčová slova

OpenGL, počítačová grafika, GLUT, načítání modelů, stencil buffer, akumulační buffer, motion blur, částicové systémy, mapování textur, bitmapové fonty, přehrávání AVI videa, detekce kolizí, fyzikální simulace

Abstract

Work is focused on computer graphics and game developing. It describes physical model of a car behaviour and implements it in form of a racing game. Game loads 3ds models and includes many interesting visual techniques. Theoretical basics are discussed in the work. Besides the program itself, a tutorial describing how to programme this is also an output of the work.

Keywords

OpenGL, computer graphics, GLUT, model loading, stencil buffer, accumulation buffer, motion blur, particle engine, playing AVI files, texture mapping, bitmap fonts, collision detection, physical simulation

Citace

Radovan Čapek: Jednoduchá hra v OpenGL. Brno, 2008, bakalářská práce, FIT VUT v Brně.

Jednoduchá hra v OpenGL

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Miroslava Švuba. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jméno Příjmení
Datum

Poděkování

Rád bych poděkoval Ing. Miroslavu Švubovi za pomoc při vypracování této práce a Bc. Janu Beránkovi za pomoc při vytváření 3ds modelu okruhu.

Obsah

1 Úvod

Každá počítačová hra se skládá z virtuálního světa nebo prostředí, do kterého může hráč pomocí komponent připojených k počítači (myš, klávesnice, joystick, joypad a další) vstoupit a nějakým způsobem ovlivňovat dění takového virtuálního prostředí, tzn. že k interakci dochází na základě zásahu hráče do spuštěného programu hry, který byl pro takový zásah vyvíjen. Obvykle se jedná o určitý úkol, který musí hráč splnit v nějakém časovém limitu, či porazit počítačový program v simulaci nějakého boje, vyhrát závodní simulaci, a nebo dostát úplně jiného cíle závislého od námětu a žánru i záměru vývojářů.

Vyvstává otázka, proč vlastně hry vyvíjet. Dnes už je ve většině případů vývoj počítačových her věcí komerce. Hry lidi baví a lidé jsou za ně ochotni zaplatit nemalý peníz. Jedná se nový fenomén v zábavě srovnatelný například s televizí a filmy. Kromě funkce zábavní a relaxační, mívají hry často výukový charakter, kdy se zábavnou formou snaží hráči podat nějaké informace.

Tvorba počítačové hry je zajímavý počin také z hlediska programování. Obsahuje většinou velké množství aplikovaných technik počítačové grafiky a často i základy fyzikálních simulací a umělé inteligence. Výstupem této práce by měla být open sourcová hra demonstrující mnoho z těchto technik, a tutoriál, ve kterém se zájemce dočte, jak tyto techniky naprogramovat s využitím OpenGL, a zprostředkovat mu tak zajímavá informace z oblasti počítačové grafiky a simulací.

Ve druhé kapitole práce je o počítačové grafice obecné pojednání. Kapitola tři obsahuje nutnou teorii k počítačové simulaci. Následuje popis návrhu a implementace aplikace.

2 Počítačová grafika

Počítačová grafika (Computer graphics - CG) [1] [4] je součástí oboru Informatika, která se zabývá analýzou (interpretací) nebo tvorbou (syntézou, generováním) grafické obrazové informace. Tento obor můžeme dělit na několik oblastí: 3D rendering v reálném čase (často využívaný v počítačových hrách), počítačová animace, video, střih speciálních efektů (často využívané ve filmu a televizi), editování obrázků a 3D modelování (například pro inženýrské nebo lékařské účely). Zpočátku se počítačová grafika rozvíjela kvůli akademickým zájmům podporovaným vládou a armádou, později však začala pronikat do filmu a televize, kde se osvědčila jako konkurenceschopná náhrada za tradiční speciální efekty a animační techniky, v důsledku toho i komerční firmy začaly přispívat k vývoji na tomto poli.

2.1 Rastrová grafika

Rastrová grafika je způsob popisu (uložení, definice) zpracovávané a zobrazované informace ve formě rastrové matice, diskrétně. Tento popis dat získáme: manuálně, syntézou (generováním nebo převodem z jiného popisu) nebo snímáním (kamerou atd.). Jeden prvek matice nazýváme pixel. Každý pixel nese specifické informace, například o jasu, barvě, průhlednosti bodu, nebo kombinaci těchto hodnot. Obrázek v rastrové grafice má omezené rozlišení, které se udává počtem řádek a sloupců.

2.2 Vektorová grafika

Vektorová grafika je způsob popisu (uložení, definice) zpracovávané a zobrazované informace ve formě skupiny vektorových entit (úsečky, kružnice, křivky, polygony, atd.), analyticky. Tento popis dat získáme: manuálně nebo syntézou (generováním nebo převodem z jiného popisu). Výhodou vektorové grafiky je libovolné zmenšování nebo zvětšování obrázku bez ztráty kvality, možnost pracovat s každým objektem v obrázku odděleně a také to, že výsledná velikost obrázku je obvykle mnohem menší než u rastrové grafiky. Každé zobrazované těleso se skládá z polygonů. Pro zobrazení se tyto tvary stínují, texturují a rasterizují.

2.2.1 Stínování

Při použití světel musíme definovat normálu povrchu. Je to přímka kolmá na daný podprostor, vektor vycházející ze středu polygonu v 90-ti stupňovém úhlu. Výsledné osvětlení se poté spočítá jako skalární součin dvou jednotkových vektorů, vektoru světla a normály polygonu. Tím dostaneme hodnotu od 0.0 do 1.0, která ovlivní barvu daného polygonu.

Aplikací tohoto postupu na každý polygon modelu dostane osvětlení scény, které je nazýváno Flat fading – plošné stínování. Tato metoda se však dnes již téměř nepoužívá, protože rozdíly v osvětlení polygonů jsou příliš viditelné.

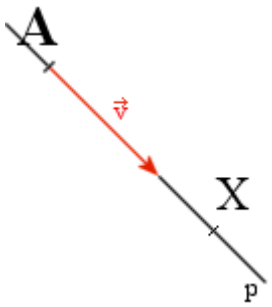
Mnohem realističtější a také náročnější metoda je *Gouraudhovo stínování*, nazývané též *Smooth shading* – plynulé stínování. Hlavním rozdílem je, že místo polygonových normál jsou pro výpočet osvětlení použity vertexové normály. Samozřejmě bod nemůže mít normálu, vertexové normály jsou počítány jako průměrné hodnoty normál polygonů, které mají za vrchol daný vertex[5].

Nejkvalitnější výsledky stínování podává metoda známá jako Phong shading – Phongovo stínování. Při Phongově stínování se obdobně spočítají normálové vektory vrcholů stěny, ale následně se pro každý vrchol stěny získá interpolací přímo normálový vektor a pro každý bod se tedy počítá barva osvětlovacím modelem zvlášť. Metoda zohledňuje zakřivení povrchu objektů a podává velmi kvalitní, realistická zobrazení. Cenou za to, je náročná implementace a vysoké nároky na výkon[6].

2.2.2 Vektorové počty

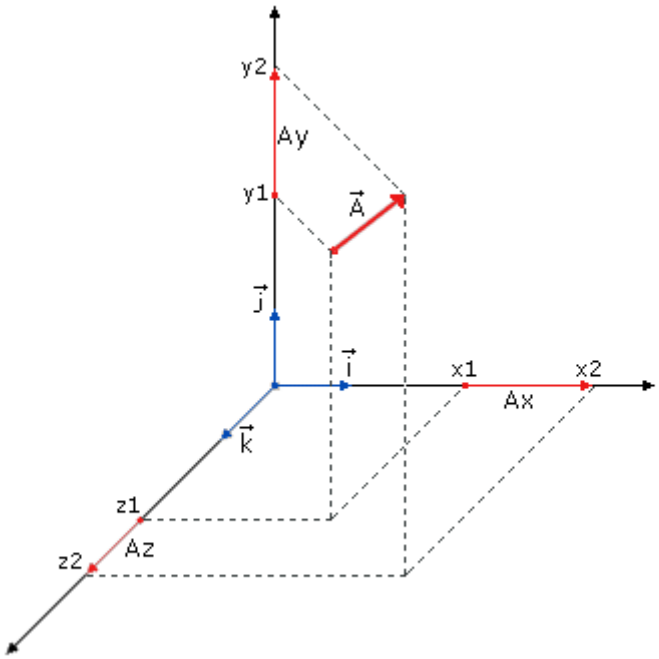
2.2.2.1 Vyjádření vektoru

V prostoru můžeme přímku vyjádřit pouze jedním způsobem, a to parametricky. K parametrickému zapsání přímky potřebujeme dva údaje. Prvním je výchozí bod, označme si jej **A** a směrový vektor **v**. Zápis bude vypadat takto : **X=A + v*t** Přičemž **X** je libovolný bod přímky a **t** je parametr, kterým násobíme směrový vektor **v**, abychom se z bodu **A** dostali do **X**.



Obrázek 2.1: Vyjádření vektoru

Vektor představuje ve fyzice a vektorovém počtu veličinu, která má kromě velikosti i směr. Tím se liší od obyčejného čísla, neboli skaláru, které má pouze velikost. V matematice je někdy definován vektor jako uspořádaná *n*-tice prvků (typicky čísel), označovaných jako složky (též komponenty) vektoru.



Obrázek 2.2: rozklad vektoru do os

Na obrázku 2.2 vidíme, že vektor A lze rozdělit do tří os. Pak je možné ho zapsat takto:

$\vec{A} = [(x2 - x1), (y2 - y1), (z2 - z1)]$ (1)

Poté:

$\vec{A} = (Ax, Ay, Az)$ (2)

Tento tvar se nazývá Kartézská reprezentace vektoru. Další možností jak reprezentovat vektor je:

$\vec{A} = Ax\,\vec{\imath} + Ay\,\vec{\jmath} + Az\,\vec{k}$ (3)

Kde $\vec{\imath}$, $\vec{\jmath}$, $\vec{k}$ jsou osy vektoru. Vezme-li v potaz, že vektor je matematická veličina, která reprezentuje úsečku velikostí a směrem, můžeme pro naše potřeby předpokládat, že všechny vektory začínají v počátku souřadného systému. Za tohoto předpokladu nám stačí k vyjádření vektoru pouze cílový bod. Struktura vektoru proto vypadá následovně:

```
typedef struct
{
    float x,y,z;
} Vector, p3d_type, *p3d_ptr_type;
```

2.2.2.2 Operace nad vektory

Pro výpočet normál polygonů (a následně vertexů) jsou některé operace nad vektory. Napřed bude nutné normalizovat vektory na jednotkové. K tomu je potřeba nejdříve znát velikost vektoru. Rozložíme-li vektor do os, zjistíme, že pro výpočet jeho velikosti můžeme použít Pythagorovu větu.

Velikost vektoru $= \sqrt[2]{Ax * Ax + Ay * Ay + Az * Az}$

Vektor pak normalizujeme na jednotkový tím, že každou jeho složku vydělíme jeho délkou. Máme jednotkové vektory, nad nimi pak bude provádět operace: sčítání, odčítání, skalární součin a vektorový součin.

Pokud použijeme zápis vektoru (3), můžeme součet vektorů zapsat analyticky jako:

$$\vec{A} + \vec{B} = (Ax + Bx, Ay + By, Az + Bz)$$

Rozdíl vektorů je pak:

$$\vec{A} - \vec{B} = (Ax - Bx, Ay - By, Az - Bz)$$

Skalární součin dvou vektorů se počítá jako:

$$\vec{A}\ dot\ \vec{B} = \vec{A} * \vec{B} * cos(\alpha)$$

Geometrický přístup se ale nehodí pro programování, proto vyjádříme rovnici analyticky.

Z geometrické rovnice vyplývá, že pro osy vektoru platí:

$\vec{\imath}\ dot\ \vec{\imath} = \vec{\jmath}\ dot\ \vec{\jmath} = \vec{k}\ dot\ \vec{k} = 1$ (4)

$\vec{\imath}\ dot\ \vec{\jmath} = \vec{\jmath}\ dot\ \vec{k} = \vec{k}\ dot\ \vec{\imath} = 0$ (5)

Podle definice (3) vektory zapíšeme takto:

$\vec{A} = Ax\,\vec{\imath} + Ay\,\vec{\jmath} + Az\,\vec{k}$

$\vec{B} = Bx\,\vec{\imath} + By\,\vec{\jmath} + Bz\,\vec{k}$

Rovnice pro skalární součin poté bude vypadat následovně:

$\vec{A}\ dot\ \vec{B} = (Ax\,\vec{\imath} + Ay\,\vec{\jmath} + Az\,\vec{k})\ dot\ (Bx\,\vec{\imath} + By\,\vec{\jmath} + Bz\,\vec{k})$

Použitím (4) a (5) můžeme nyní zapsat skalární součin analytickým způsobem:

$$\vec{A}\ dot\ \vec{B} = Ax * Bx + Ay * By + Az * Bz$$

Nakonec vypočítáme analytickou rovnici pro vektorový součin:

$i\ x\ i = j\ x\ j = k\ x\ k = 0$ (6)

$i\ x\ j = k\ j\ x\ k = i\ k\ x\ i = j$ (7)

Opět vyjádříme vektory podle definice (3):

$\vec{A} = Ax\,\vec{\imath} + Ay\,\vec{\jmath} + Az\,\vec{k}$

$\vec{B} = Bx\,\vec{\imath} + By\,\vec{\jmath} + Bz\,\vec{k}$

$\vec{A} \times \vec{B} = (Ax\,\vec{\imath} + Ay\,\vec{\jmath} + Az\,\vec{k}) \times (Bx\,\vec{\imath} + By\,\vec{\jmath} + Bz\,\vec{k})$

Po aplikaci kroků (6) a (7) můžeme vektorový součin vyjádřit analyticky:

$$A \times B = (AyBz - AzBy)i + (AzBx - AxBz)j + (AxBy - AyBx)k$$

Aby osvětlení fungovalo správně, je ještě potřeba nastavit správné atributy světel a materiálů. Operace nad vektory se kromě výpočtu normál pro osvětlení používají především při fyzikální reprezentaci pohybu objektů, ale i pro výpočet kolizí a mnoha dalších výpočtech.

2.2.3 Texturování

Textura [4] je popis detailní struktury povrchu objektu, nezávislý na jeho geometrii. Jeden vzorek textury nazýváme texel. Proces nanášení textur na geometricky definovaný povrch objektu nazýváme texturování. Z hlediska vizuálního vnímání textura pomáhá vzájemně odlišovat a rozpoznávat jednotlivé objekty. Z hlediska zobrazování počítačových 3D modelu objektu umožňují textury dosáhnout řádově vyššího stupně realističnosti výsledného obrazu.

Textury mohou na povrchu objektu popisovat (mapovat) různé vlastnosti:
Barva povrchu – nejčastější způsob použití textur. Textura nanese na povrch objektu barevnou strukturu v podobě RGB informace.
Světelné vlastnosti povrchu – difuze a odrazivosti. Mění charakter odrazu světla na povrchu objektu při výpočtu osvětlení (Phongův osvětlovací model. viz. kapitola 2.2.1). Simuluje lokálně ušpiněný (matný) nebo vyleštěný povrch, atd.
Průhlednost – nerovnoměrně průhledné objekty, prolínání vlastností definovaných
různými texturami přes sebe, oříznutí geometrie objektu nastavením úplné průhlednosti vybraným částem atd. Je tak možné nanesením vhodné textury na jednoduchou geometrii (např. obdélník, hranol, kouli atd.) modelovat složité objekty (listí, trávu, atd.).
Modifikace normály – Bump textury. Provádí modifikaci normály podle hodnoty gradientu v textuře. Vzniká čistě optický efekt hrbolatého povrchu.
Změna geometrie – Displacement textury. Provádí posun povrchových bodu ve směru normály podle hodnoty v textuře. Dochází ke skutečné lokální změně geometrie.
Hypertextura – určuje optické vlastnosti nad povrchem objektu, vhodné pro zobrazení objektu typu ohně, vlasu, trávy, atd.
Osvětlení – Light textury. Realizují off-line difuzní statické osvětlení povrchu.
Zrcadlení okolí – Environment textury. Nanáší na lesklý povrch obraz okolí.

2.2.4 Rasterizace

Rasterizace je proces, při kterém se vektorově definovaná grafika konvertuje na rastrově definované obrazy s ohledem na maximální možnou kvalitu a rychlost zobrazení. Protože složitější objekty jsou jen skládankou jednodušších objektů, potřebujeme mít k dispozici algoritmy na výpočet polohy bodů jednoduchých objektů jako úsečky, kružnice, elipsy, oblouků, atd.

2.3 Maticové transformace

Popis těles vektorové grafiky je založen na vertexech, uzlových bodech. Při manipulaci s tělesem v prostoru je nutné manipulovat právě s těmito body. Tělesa se pak mohou na obrazovce zobrazit v libovolném posunu, natočení či měřítku. Tyto manipulace jsou nazývány *transformacemi*. K provádění transformací je třeba na každý vrchol tělesa aplikovat určitou obdobu vzorce. K vykreslení transformovaného tvaru se pak použijí nově modifikované vrcholy [2]. Mezi transformace posunutí, otáčení, změna měřítka, zkosení.

Počítání těchto transformací za pomocí volání funkcí by bylo značně časové náročné (geometrické transformace vektorových objektu jsou jednou z nejčastěji používaných operací v současné počítačové grafice), proto se začaly k transformování objektů používat matice. *Matice* je jednoduše řečeno tabulka čísel uspořádaných do řad a sloupců. V 2D počítačové grafice jsou používány matice 3x3, v třídimenzionálním prostoru matice 4x4.

2.3.1 Posunutí

Při posunutí dochází ke změně umístění vrcholů v souřadnicovém systému. Posunová matice T pro 3D objekty vypadá následovně:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ d_x & d_y & d_z & 1 \end{bmatrix}$$

2.3.2 Otáčení

Při rotaci trojrozměrného objektu dochází ke změně natočení zobrazeného objektu. Při otáčení v 2D prostoru lze tvar otáčet pouze kolem jedné osy, při otáčení v 3D lze tělesa otáčet kolem všech tří os. Neexistuje však žádná matice umožňující provádět rotace kolem všech tří os současně, proto se provádí samostatně a každá osa má vlastní matici otáčení R:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & \cos\alpha & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.3.3 Změna měřítka

Při změně velikosti násobíme vrcholy 3D objektu faktorem změny, který zvětšuje nebo zmenšuje rozměry objektu.

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.3.4 Zkosení

Podobně jako u rotační transformace ve 3D je transformace zkosení ve 3D rozdělena na tři různé operace, podle směru, ve kterých zkosení probíhá. Máme potom tři transformační matice SHYZ, SHXZ, SHXY pro zkosení ve směrech YZ, XZ a XY:

$$S_{HYZ} = \begin{bmatrix} 1 & S_{hy} & S_{hz} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S_{HYZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ S_{hx} & 1 & S_{hz} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S_{HYZ} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ S_{hx} & S_{hy} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.3.5 Skládání transformací

V praxi je však většinou potřeba aplikovat složitější transformace, ty dostaneme složením základních transformací ve správném pořadí. Skládání dvou matic se realizuje jejich vzájemným vynásobením, výsledek pak je finální maticí.

Obrázek 2.3: Skládání transformací

3 Fyzikální model

Většinu her a počítačových simulací spojuje snaha o vytvoření věrné kopie reálného světa, kde se chování a interakce objektů řídí pravidly popsanými fyzikálními zákony. Postihnutí všech fyzikálních zákonů však není možné a navíc by to bylo výpočetně nesmírně náročné. Simulaci je tedy třeba vhodně generalizovat. První krokem je vybrání vhodných majoritních fyzikálních činitelů. Počítáme-li například odraz světla od povrchu podložky, je zbytečné započítat lom způsobený okolím, pokud je okolí paprsku homogenní nebo působní gravitace na paprsek. Dalším krokem je pak zjednodušit fyzikální zákony, aby jejich výpočet nebyl příliš náročný, ale zároveň dostatečně věrně odpovídal skutečnosti. Většinou platí, že čím je simulace přesnější, tím je výpočet náročnější. V neposlední řadě je dobré aplikovat fyziku pouze na vhodné objekty.

3.1 Simulace fyzikálního modelu vozidla

Klíčem ke zjednodušení fyzikální modelu auta je počítat odděleně složku síly působící ve směru pohybu vozidla a síly působící kolmo na tento směr. Složení sil volíme podle toho, jak má být simulace reálná. Základní síly čelního pohybu, které však měly být použity, jsou: hnací síla kol, brzdná síla, třecí síla valivého odporu, brzdná síla odporu vzduchu. Síly kolmé k silám čelního pohybu umožňují zatáčení auta [9] [10].

3.1.1 Hnací síla

Hnací síla je síla vyvíjená motorem, která je skrze kola působením kroutícího momentu přenášena na podložku a odpor podložky následně způsobí pohyb vozidla.

Předpokládáme-li, že velikost hnací síly se rovná velikosti proměnné Engineforce, která je kontrolována přímo řidičem, můžeme zapsat velkost hnací síly:

$$\boldsymbol{F}_{traction} = \boldsymbol{u} * Engineforce,$$

$$kde\ \boldsymbol{u}\ je\ jednotkový\ vektor\ ve\ směru\ natočení\ vozidla.$$

Pokud by hnací síla byla jedinou silou působící na vozidlo, vozidlo by donekonečna zrychlovalo. Proti pohybu vozidla však působí odporové síly. První, a pravděpodobně nejdůležitější, z nich je odpor vzduchu. Tato síla je tak důležitá proto, že její velikost vzrůstá kvadraticky. Tak se tato zpočátku malá odporová síla ve vysokých rychlostech projeví nejvíce.

$$\boldsymbol{F}_{drag} = -Cdrag * \boldsymbol{v} * |\boldsymbol{v}|$$

$$kde\ Cdrag\ je\ konstanta, \boldsymbol{v}\ je\ vektor\ rychlosti\ a\ |\boldsymbol{v}| označuje\ velkost\ vektoru\ \boldsymbol{v}$$

Velikost rychlostního vektoru nám dává skalární veličinu, kterou známe jako rychlost.

Třecí síla je způsobena třením mezi pneumatikami a povrchem vozovky. Její velikost dostaneme následovně:

$$\boldsymbol{F}_{rr} = -Crr * \boldsymbol{v}$$
$$kde\ Crr\ je\ konstanta\ a\ \boldsymbol{v}\ je\ vektor\ rychlosti.$$

Výslednou sílu čelního pohybu dostane součtem uvedených sil:

$$\boldsymbol{F}_{long} = \boldsymbol{F}_{traction} + \boldsymbol{F}_{drag} + \boldsymbol{F}_{rr}$$

Odporové síly mají opačný směr než síla hnací.
Rovnici pro akceleraci vozidla dostaneme z druhého Newtonova zákona:

$$\boldsymbol{a} = \boldsymbol{F} / M \quad \left[\frac{m}{s} = N/kg\right]$$

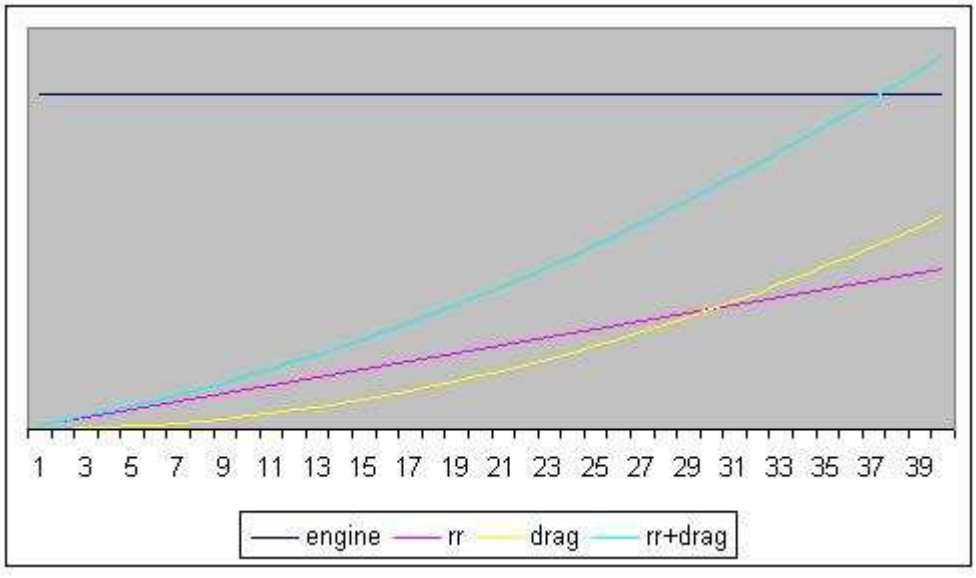
Integrací vektoru *zrychlení podle času* je vektor okamžité rychlosti

$$\boldsymbol{v} = \boldsymbol{v} + \Delta t * \boldsymbol{a},$$
$$kde\ \Delta t\ je\ přírůstek\ času.$$

Integrací vektoru okamžité rychlosti podle času je vypočtena dráha (vzdálenost) od bodu na počátku integrace

$$\boldsymbol{p} = \boldsymbol{p} + \Delta t * \boldsymbol{v}$$

Užitím těchto tří sil můžeme poměrně relativně simulovat zrychlení. Jejich působení nám také udává maximální možnou rychlost vozidla, jak je vidět z obrázku 3.1.



Obrázek 3.1: Působení dopředných sil

3.1.2 Brzdná síla

Při brzdění je hnací síla nahrazena silou brzdnou, která působí v opačném směru. Výslednicí sil je potom součet všech tří dopředných sil.

$$\boldsymbol{F}_{long} = \boldsymbol{F}_{braking} + \boldsymbol{F}_{drag} + \boldsymbol{F}_{rr}$$

Jednoduchý model brzdné síly:

$$\boldsymbol{F}_{braking} = -\boldsymbol{u} * Cbraking$$

3.1.3 Přenos váhy

Důležitým jevem při zrychlení nebo zpomalení je odlehčení nápravy. Při brzdění předek vozidla klesne, zatímco při akceleraci se auto mírně zakloní. Tento jev je důležitý hned ze dvou důvodů. Zaprvé je to vizuální odezva na řidičovy akce a udělá tak simulaci realističtější. A zadruhé rozložení váhy dramaticky ovlivní maximální hnací sílu působící na kolo, protože tření (jehož důsledkem je dopředný pohyb) má limit závislý na zatížení kola.

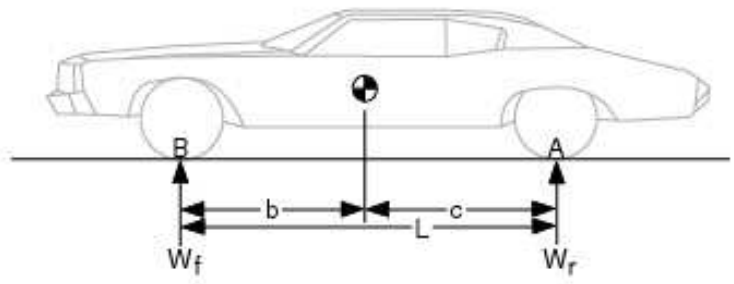
$$\boldsymbol{F}_{\max} = mu * W$$
$$kde\ mu\ je\ koeficient\ přilnavosti\ pneumatik.$$

Pokud se vozidlo nehýbe, je váha (W, což se rovná M*g) rozdělena mezi přední a zadní nápravu podle vzdálenosti od těžiště:

$$Wf = \left(\frac{c}{L}\right) * W$$
$$Wr = \left(\frac{b}{L}\right) * W$$

$kde\ b\ je\ vzdálenost\ přední\ nápravy\ od\ těžiště, c\ je\ vzdálenost\ zadní\ nápravy\ od\ těžiště$

$a\ L\ je\ rozvor\ kol."$



Obrázek 3.2: Přenos váhy

Pokud vozidlo zrychluje nebo zpomaluje, váha na přední (W_f) a zadní nápravě (W_r) se vypočítá následovně:

$$Wf = (c/L) * W - (h/L) * M * a$$
$$Wr = (b/L) * W + (h/L) * M * a,$$
$$kde\ h\ výška\ těžiště, M\ je\ hmotnost\ vozidla\ a\ a\ je\ akcelerce.$$

3.1.4 Kroutící moment motoru

Kroutící moment vyjadřuje působní síly na bod vzdálený od osy otáčení. Kroutící moment je spolu s výkonem a maximálními otáčkami za minutu důležitým parametrem spalovacích motorů (týká se ale všech rotačních pohonných systémů, tedy všech točivých motorů všech typů a možných konstrukčních provedení).

Přepočet kroutícího momentu a otáček motoru na výkon:

$$hp = torque * rpm / 5252$$

Kroutící moment je přes převodovku a diferenciál převeden na kola vozidla. Převodový stupeň znásobí kroutící moment v závislosti na převodovém poměru. Je třeba započítat i to, že 30% se ztratí ve formě tepla. Sílu, kterou působí kolo na vozovku, vypočteme z krouticího moment působícího na hnací nápravu tím, že ho vydělíme poloměrem kola. Rovnice pro převod z kroutícího momentu motoru na hnací sílu:

$$\boldsymbol{F}_{drive} = \boldsymbol{u} * Tengine * xg * xd * n / Rw$$
$$kde\ je\ jednotkový\ vektor\ ve\ směru\ orientace\ vozidla,$$
$$Tengine\ je\ kroutící\ moment\ motoru\ daný\ počtem\ otáček,$$
$$xg\ je\ převodový\ poměr,$$
$$xd\ je\ poměr\ diferenciálu,$$
$$n\ je\ efektivita\ převodu\ a$$
$$Rw\ je\ poloměr\ kola.$$

3.1.5 Skluzový poměr a hnací síla

Výpočet úhlové rychlosti kola jen z rychlosti by bylo možné pouze v případě, že by nedocházelo k žádnému skluzu mezi vozovkou a kolem. Ve skutečnosti se ale hnací kola při záběru otáčí o něco rychleji. Běžně bychom mohli úhlovou rychlost spočítat vydělením rychlosti auta dvěma π. Avšak u pneumatik hnací nápravy dochází k prokluzování v závislosti na povrchu vozovky. Skluzový poměr vyjadřuje rovnice:

$$\sigma = \frac{\omega_w R_w - \boldsymbol{v}_{long}}{|\boldsymbol{v}_{long}|}$$

kde

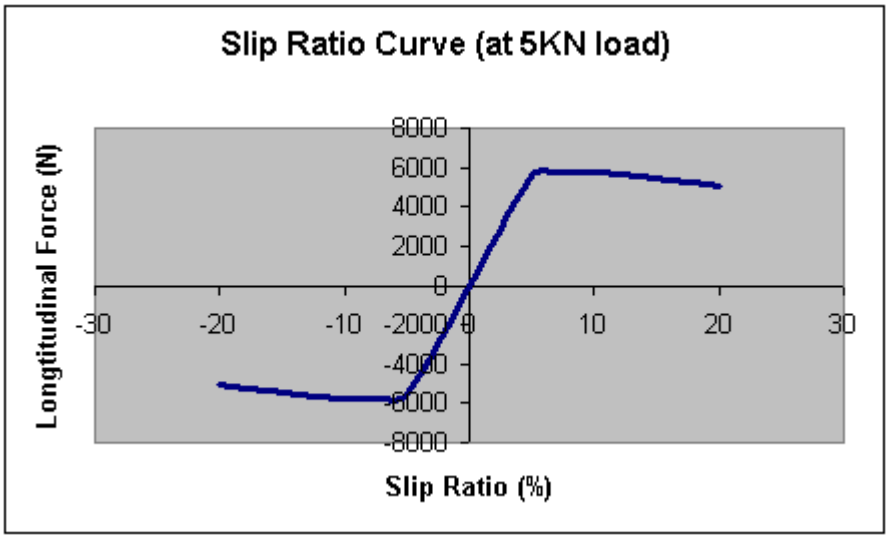
ω_w je úhlová $rychlost$ $(v\ rad/s)$

R_w je $poloměr$ $kola$ $(v\ m)$

$\boldsymbol{v}_{long}$ je $rychlost$ $vozidla$ $(v\ m/s)$

Pro volně se otáčející kola je skluzový poměr roven nule. Při brzdění se zablokovanými koly je roven -1. Pokud vozidlo zrychluje je skluzový poměr kladný.

Přibližný vztah mezi dopřednou silou a skluzným poměrem je popsán křivkou v grafu 3.3:



Obrázek 3.3: Křivka skluzového poměru

Z grafu je vidět, že síla je nejlépe přenášena při 6% skluzovém poměru. Přesná křivka je závislá na pneumatikách, povrchu, teplotě atd.

Kola tedy nejlépe zabírají při mírném skluzovém poměru. Za optimální hodnotou skluzového poměru účinnost přenosu síly klesá. Lepší akceleraci a držení stopy mají tedy kola při mírném skluzu, než pokud ke skluzu vůbec nedochází.

Abychom dostali z jednotkového vektoru dopředné síly současnou dopřednou sílu, vynásobíme ji hmotností působící na kolo:

$$F_{long} = F_{n,long} * F_z$$

kde $F_{n,long}$ je $normalizovaný$ $vektor$ $dopředné$ $síly$ pro $získání$ $skluzového$ $poměru$

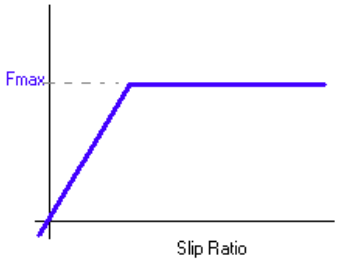
a F_z je $zatížení$ $kola.$

Pro jednoduchou simulaci pak můžeme výslednou sílu aproximovat lineárním vztahem:

$$F_{long} = Ct * slip\ ratio$$

kde Ct $představuje$ $konstantu$ $přilnavosti,$ $která$ $odpovídá$ $naklonění$ $křivky$ $původního$ $grafu$

Po dosažení optimální hodnoty skluzového poměru se hodnota síly už nemění. Dostaneme tak funkci, jejíž průběh se pro potřeby simulace blíží původnímu grafu.



Obrázek 3.4: Zjednodušená křivka skluzového poměru

3.1.6 Přenos točivého momentu na kola

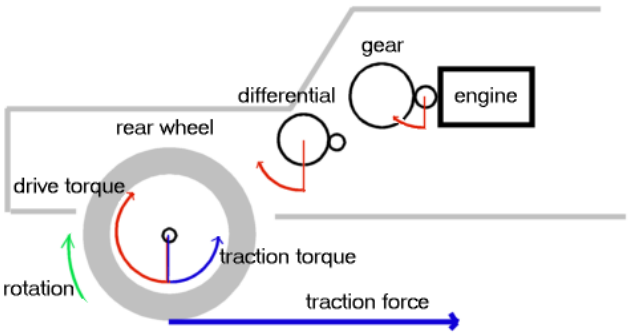
Hnací síla vzniká jako odporová síla, kterou působí povrch vozovky na povrch pneumatik. Podobně vzniká síla při přenosu točivého momentu na kola:

$$traction\ torque\ =\ traction\ force\ *\ wheel\ radius$$

Točivý moment působící na zadní kolo nápravy je součtem následujících točivých momentů:

$$total\ torque\ =\ drive\ torque\ +\ traction\ torques\ from\ both\ wheels\ +\ brake\ torque$$

Točivý moment kol (drive torque) má obvykle opačné znaménko než hnací točivý moment (traction torques) a brzdící točivý moment kol (brake torque). Pokud řidič nebrzdí je brzdící točivý moment nulový. Působení uvedených momentů je zobrazeno na obrázku 3.5.:



Obrázek 3.5: Působní točivých momentů

Působení točivého momentu způsobuje změnu úhlové rychlosti kol, stejně jako při aplikování síly na hmotný bod dochází ke zrychlení.

$$angular\ acceleration\ =\ total\ torque\ /\ rear\ wheel\ inertia$$

Pro setrvačnost (inertia) válce platí vztah:

$$inertia\ of\ a\ cylinder\ =\ Mass\ *\ radius^2\ /\ 2$$

Při nenulovém úhlovém zrychlení se v čase mění úhlová rychlost kol. Pro její výpočet provedeme krok numerické integrace vztahem:

$$rear\ wheel\ angular\ velocity\ +=\ rear\ wheel\ angular\ acceleration\ *\ time\ step$$

Časový úsek (time step) vstupuje do fyzikální funkce jako parametr.

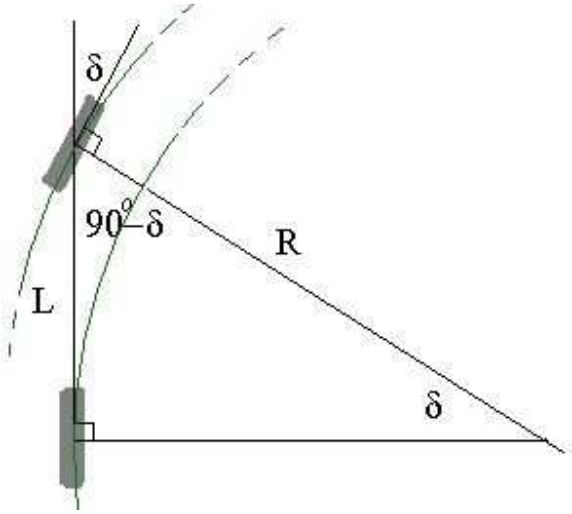
3.1.7 Zatáčení

Při simulace zatáčení vozidla musíme vzít v úvahu rozlišnost mezi zatáčením v nízké a ve vysoké rychlosti. Při nízkých rychlostech se kola pohybují ve směru, kam jsou natočeny. Taková simulace se provádí za pomoci pravidel geometrie a kinetiky. Není potřeba vzít v úvahu setrvačné síly a hmotnost vozidla. Jinými slovy, jedná se o problém kinematiky, ne dynamiky.

Ve vyšších rychlostech kola mohou být natočeny v jednom směru, ale stále setrvávat ve směru původního pohybu. Kola se tak nepohybují ve směru své orientace. Počítat je třeba se složkou rychlosti, která působí v pravém úhlu k natočení kol. Kola jsou uzpůsobena k tomu, aby se lehce pohybovala ve směru natočení, pohyb v jiném směru vyžaduje velké působení síly k překonání třecích sil, proto tyto síly vznikají jen ve vyšších rychlostech a pak je třeba je započítat.

3.1.8 Zatáčení v nízkých rychlostech

V nízkých rychlostech se kola valí, ale neklouzají stranou. Pokud jsou přední kola natočena v úhlu delta a vozidlo se pohybuje konstantní rychlostí, pak by se vozidlo mělo pohybovat po kružnici. Ze středu zadního a natočeného předního kola je možné si představit polopřímky směřující do zatáčky, jejich průsečík je pak středem kružnice, po které se vozidlo pohybuje. Poloměr kružnice může být znázorněn geometricky diagramem 3.6:



Obrázek 3.6: Poloměr kružnice otáčení

Vzdálenost mezi přední a zadní nápravou se nazývá rozvor a označuje se L. Poloměr kružnice opisované předním kolem značíme R. Z diagramu je vidět, že vrchol vzniklého trojúhelníku bude mít u středu otáčení úhel delta. Sinus tohoto úhlu se pak spočítá jako rozvor děleno průměrem kružnice, pak platí:

$$\sin(\delta) = \frac{L}{R} \Leftrightarrow R = \frac{L}{\sin(\delta)}$$

Pokud je úhel zatáčení nulový, průměr kružnice je pak nekonečny, vozidlo se tedy pohybuje po přímce. Dalším krokem je výpočet úhlové rychlosti, tedy rychlosti jakou se vozidlo otáčí. Úhlová

rychlost se obvykle značí řeckým písmenem omega (ω) a její jednotkou jsou radiány za sekundu. Rovnice pro úhlovou rychlost je dána vztahem:

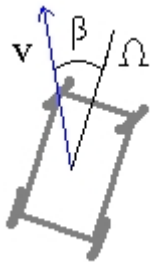
$$\omega = \frac{v}{R}$$

K výpočtu zatáčení v nízkých rychlostech je toto vše, co potřebujeme znát. Úhel natočení kol je dán uživatelským vstupem. Rychlost auta je určena jako pro případ, kdy se vozidlo pohybuje po přímce. Z ní spočítáme poloměr kružnice a úhlovou rychlost. Úhlová rychlost je použita pro změnu orientace vozidla. Rychlost vozidla je otáčením nedotčena, její vektor se pouze otáčí spolu s orientací vozidla.

3.1.9 Zatáčení ve vysokých rychlostech

Ve vysokých rychlostech již neplatí, že auto se pohybuje tím směrem, kam směřují kola. Tělo vozidla má určitou hmotnost, a tak v důsledku setrvačnosti trvá, než vozidlo zareaguje na řízení. Vozidlo může mít nenulovou úhlovou rychlost. Stejně jako u klasické rychlosti chvíli trvá její nárůst a zpomalení. Tato doba je dána úhlovým zrychlením, které je závislé na točivém momentu a setrvačnosti (což jsou rotační ekvivalenty k síle a hmotnosti).

Vozidlo se také vždy nemusí pohybovat ve směru, kam je orientováno. Předek auta může směřovat do určitého místa, ale pohybovat se může jinam. Úhel mezi orientací auta a směrem vektoru rychlosti auta je označován jako úhel bočního smyku vozidla (beta).



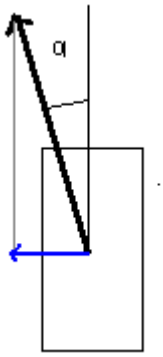
Obrázek 3.7: Úhel bočního smyku

Pokud se podíváme na situaci zatáčení ve vysokých rychlostech z pohledu kol, budeme potřebovat spočítat boční pohyb pneumatik. Jak bylo již zmíněno, kola kladou malý odpor při valivém pohybu vpřed a vzad, zato mají velkou rezistenci proti bočnímu pohybu. Síla působící na boční pohyb pneumatik závisí na hmotnosti vozidla a na pneumatikách samotných. Při malém úhlu smyku je vztah mezi silou bočního pohybu a skluzným úhlem lineární, matematicky zapsáno:

$$\mathrm{F_{lateral}} = \mathrm{C_a} * \alpha$$

$$kde\ C_a\ je\ konstanta\ tuhosti$$

Vektor rychlosti svírá úhel alfa se směrem otáčení kol. Vektor rychlosti můžeme rozdělit na dvě navzájem kolmé složky. Jak je z obrázku 3.8 zřejmé, má dopředná složka vektoru rychlosti velikost $cos(\alpha) * v$. Tato složka má stejný směr jako je směr orientace kol. Kolmá složka má velikost $sin(\alpha) * v$ a působí jako odporová síla ve směru proti pohybu do zatáčky.



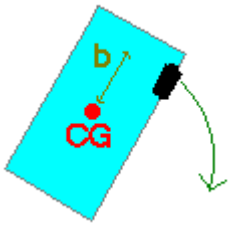
Obrázek 3.8: Rozložení vektoru rychlosti

Úhel bočního smyku auta, úhlová rotace auta okolo osy Y a úhel natočení předních kol jsou tři věci, které zásadně ovlivňují úhel smyku kol.

Úhle bočního smyku beta je rozdíl mezi natočením auta a skutečným směrem pohybu. Jeho koncept je podobný jako u bočního smyku kola a můžeme ho spočítat ze vztahu:

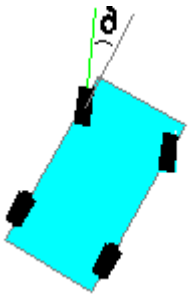
$$\beta = tan^{-1}\left(\frac{v_x}{v_y}\right)$$

Pokud se auto otáčí okolo svého těžiště úhlovou rychlostí omega, znamená to, že přední kola opisují kružnici stejnou úhlovou rychlostí. Když auto opíše kružnici celou, přední kola urazí pohybem pro kružnici vzdálenost o velikost $2\pi b$ za čas $1/2\pi\omega$ sekund, kde b je vzdálenost přední nápravy od těžiště. Z toho plyne, že velikost kolmé složky rychlosti je ωb, pro zadní kola pak logicky $-\omega c$.



Obrázek 3.9: Otáčení vozidla kolem těžiště

Úhel natočení kol delta je úhel mezi vytočením předních kol a směrem orientace vozidla. Zadní kola nikdy žádný úhel natočení nemají, jsou vždy ve stejné rovině s vozidlem. Pro couvající vozidlo počítáme s převrácenými hodnotami úhlu natočení kol.



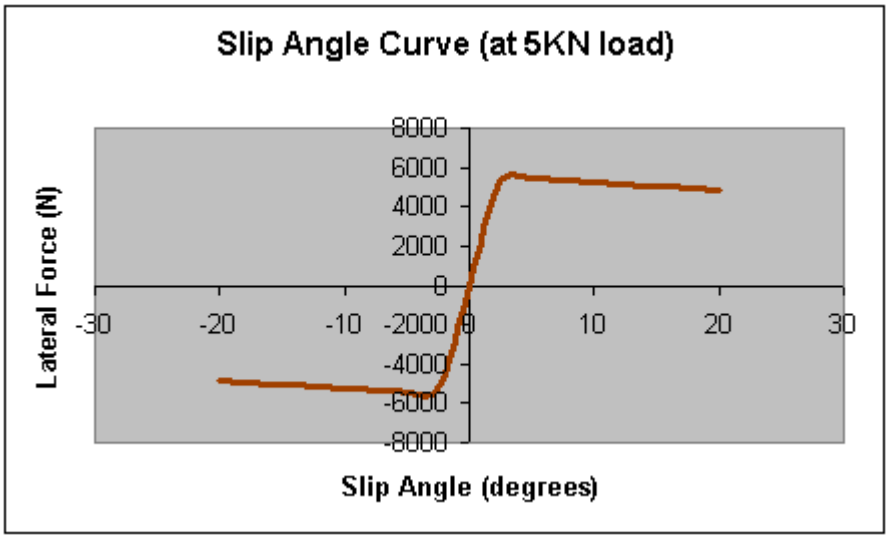
Obrázek 3.10: Orientace kol vozidla

Skluzné úhly pro přední a zadní kola jsou pak dány rovnicemi:

$$\alpha_{front} = tan^{-1}\left(\frac{v_{lat} + \omega b}{|v_{long}|}\right) - \delta\, sgn(v_{long})$$

$$\alpha_{rear} = tan^{-1}\left(\frac{v_{lat} - \omega c}{|v_{long}|}\right)$$

Funkce $sgn(number)$ vrací hodnotu 1, pokud je argument kladný, -1 pokud je záporný a nulu v případě vstupu nulové hodnoty. Kolmá síla pneumatik je funkcí úhlu skluzu. Ve skutečnosti je situace komplikovanější, skutečný graf velikosti kolmé síly pneumatik v závislosti na skluzu může znázorňovat křivka v grafu 3.11.:



Obrázek 3.11: Křivka skluzového úhlu

Graf konkrétní kolmých sil vznikající při určitém skluzovém úhlu pro specifický druh pneumatik. Předešlé dva grafy si jsou podobné, první však vyjadřuje závislost dopředné síly na čelním skluzu, druhý pak boční síly na skluzovém úhlu.

Velikost boční síly nezávisí ale jen na skluzovém úhlu, ale také na břemenu pneumatiky. V grafu 3.11. se jedná například o břemeno působící silou 5kN na jednu pneumatiku, tedy asi váhu 500kg. V případě jiné váhy by se graf změnil, ale tvar křivky by zůstal téměř stejný. Problém můžeme zjednodušit na tuto lineární závislost:

$$F_{lateral} = F_{n,lat} * F_z$$

$$kde\ F_{n,lat}\ je\ normalizovaná\ boční\ síla\ pro\ daný\ skluzový\ úhel$$

$$a\ F_z z\ je\ váha\ na\ pneumatice.$$

Pro velmi malé úhly (před vrcholem křivky) můžeme funkci zjednodušit na:

$$F_{lateral} = Ca * alpha$$

Pokud bychom chtěli aproximovat vztah mezi skluzovým úhlem a boční silou lépe, je vhodné použít tzv. Pacejkovu magickou formuli, pojmenovanou podle profesora Pacejky z Delftské university. Jeho model využívají fyzici k simulaci chování pneumatik. Užitím správných konstant

dosahuje tento model velmi věrných výsledků. Nevýhodou modelu je, že musíme znát přesné číselné hodnoty vlastností pneumatiky.

Výsledkem působení bočních sil všech 4 kol pak je výsledná síla zatáčení a rotace kolem ypsilonové osy. Síla zatáčení je síla působící na těžiště v kolmém směru ke směru orientace auta a slouží jako dostředivá síla potřebná k opsání kružnice. Pro zadní kola můžeme velikost jejich bočních sil jednoduše k síle zatáčení přičíst, u předních kol pak ještě velikost jejich boční síly musíme vynásobit $cos(\delta)$ k zohlednění natočení kol.

$$F_{cornering} = F_{lat,rear} + cos(\delta) * F_{lat,front}$$

V našem zájmu je najít poloměr kružnice, po které se vozidlo pohybuje, vyvodit ho můžeme ze vztahu:

$$F_{centripetal} = M\, v^2 \,/\, radius$$

Boční síla také představuje točivý moment, který působí otáčení vozidla. Ovšem není možné, aby vozidlo opisovalo kružnici a přitom bylo natočeno stále jedním směrem. Proto je nutné ještě spočítat otáčení kolem osy y.

Točivý moment je síla kolmá na vzdálenost mezi bodem působení síly a středem otáčení. Jeho velikost je pak násobek působící síly a této vzdálenosti. Takže pro zadní kola je přídavek k celkovému točivému momentu $-F_{lat,rear} * c$, pro přední kola je to $cos(\delta) * F_{lat,front} * b$. Stejně jako pro sílu platí druhý Newtonův zákon $F = M * a$, platí stejný zákon i pro točivý moment a zrychlení úhlové.

$$Torque = Inertia * angular\ acceleration$$

Setrvačnost (inertia) pro pevnou karoserii je konstantní a závislá na hmotnosti a konstrukci.

4 Návrh

4.1 Nástroje programového řešení

Pro tvorbu 3D počítačové grafiky se nabízí především dvě API – OpenGL a DirectX. DirectX [3] bylo původně vyvinuto speciálně pro tvorbu her. Obsahuje tedy prakticky vše potřebné, umí si poradit s grafikou, videem, klávesnicí… Má ale také své zápory: je třeba napsat velké množství kódu, aby program vůbec něco dělal, jednotlivé verze mají problémy s kompatibilitou a hlavně DirectX jsou platformě závislé na Windows (nově i Xbox360). Proto jsem zvolil OpenGL.

V dnešní době je vyvinuto mnoho vizualizačních toolkitů pro 3D grafiku a tvorbu her. Mým cílem je však ukázat jádro tvorby počítačové grafiky. S ohledem na to jsem se snažil vybírat nízko-úrovňové prostředky. Nevyužívám proto žádných herních knihoven, ani výkonných vizualizačních toolkitů. Pro usnadnění práce s oknem a snadnější obsluhu klávesnice a myši používám pouze nadstavbovou knihovnu OpenGL GLUT.

V kapitole 4.1.1 široký rámec programovacích jazyků, ve kterých je možno OpenGL aplikace programovat. K implementaci práce jsem vybral jazyk C/C++ pro jeho rozšířenost a svižnost.

4.1.1 OpenGL

OpenGL (*Open Graphics Library*) [7] [8] je průmyslový standard specifikující multiplatformní rozhraní (API) pro tvorbu aplikací počítačové grafiky. Používá se při tvorbě počítačových her, CAD programů, aplikací virtuální reality, vědeckotechnické vizualizaci, či jiných grafických aplikací.

Základní funkcí OpenGL je vykreslování do obrazového rámce (framebufferu). Umožňuje vykreslování různých základních primitiv (bodů, úseček, mnohoúhelníků) v několika různých režimech, ale je možno také na něj pohlížet jako na stavový automat. Poskytuje nám možnost měnit vlastnosti primitiv nebo scény (např. barva, buffery, antialiasing, textury, světla, materiály, translace…) a toto nastavení zůstane zachováno, dokud ho explicitně nezměníme.

Rozhraní OpenGL je založeno na architektuře klient-server, program (klient) vydává příkazy, které grafický adaptér (server) vykonává. Díky této architektuře je například možné, aby program fyzicky běžel na jiném počítači než na tom, na kterém se příkazy vykonávají, a příkazy se předávaly prostřednictvím počítačové sítě.

OpenGL bylo postaveno jako 3D renderující systém, který může být HW akcelerován. Operace se tedy provádí buď programově (vykonává procesor CPU), nebo se předá ovladači 3D akcelerátoru (vykonává grafický procesor GPU).

Aby zůstalo platformě nezávislé neobsahuje žádné funkce pro práci s okny ani s GUI (Graphical User Interface), stejně jako neumí pracovat ani se zvukem, tiskem, myší a klávesnicí či jinými

vstupními zařízeními. Díky tomu je dnes OpenGL dostupné na většině operačních systémů (Windows, Unix, Linux, Irix, Sun). K velkému rozšíření OpenGL však nemalou měrou přispěla nejen platformní nezávislost, ale také možnost programování aplikací v mnoha jazycích jako jsou Ada, C, C++, Fortran, Java, ObjectPascal, Python, Perl , assemblery apod. OpenGL v rámci platformní nezávislosti přináší i své vlastní datové typy jako GLbyte, GLint nebo GLdouble.

4.1.2 Pohled do historie

V 80. letech bylo psaní aplikací používající grafický HW neskutečně obtížné. SW vývojáři psali API a ovladače pro každé zařízení zvlášť. To vedlo k neustálému předražování vývoje a duplicitě kódu, kdy mnozí psali něco, co už dávno před nimi napsal někdo jiný pro jiné zařízení. Na počátku 90. let byla v segmentu 3D grafických pracovních stanic firma SGI (Silicon graphics). Jejich IrisGL API se díky svému kvalitnímu návrhu a snadnému používání stalo de-facto průmyslovým standardem. Konkurenti SGI (společnosti jako IBM, HP, SUN) používali pro svůj 3D hardware konkurenční standard PHIGS, který byl považován za částečně zastaralý. Následkem toho se pozice firmy SGI na trhu neustále oslabovala s narůstajícím počtem konkurentů používajících standard PHIGS. Za této situace přichází SGI s revolučním řešením a rozhodne se proměnit IrixGL na otevřený standard. Vyústěním těchto snah bylo uvedení OpenGL standardu společností SGI, odvozeného od IrisGL.

OpenGL sjednotila přístup k HW, a přesunula odpovědnost za obsluhu HW od vývojářů aplikací směrem k výrobcům HW. Dnes se tento revoluční krok projevuje například existencí ovladače ke grafické kartě a všichni berou ovladače jako samozřejmost. Přesto je to inovace, za niž vděčíme právě vzniku OpenGL. Jednotná řeč pro všechen možný grafický HW měla velký pozitivní vliv na vývoj 3D aplikací. V roce 1992 SGI podpořila vznik uskupení *OpenGL architectural review board* (*OpenGL ARB*), tedy jakousi asociaci společností, která měla zajistit vývoj OpenGL do budoucna. Jednou z mnoha společností tohoto uskupení byl i Microsoft, který jej ovšem v roce 2003 opustil. Od roku 2006 spravuje OpenGL skupina *Khronos group*. Mezi nejznámější členy patří např. ATI technologies, AMD, Creative Labs, Intel, ID Software, Nvidia, Sony Computer Entertainement, Sun Microsystems a mnoho dalších.

4.1.3 Rozšíření OpenGL

Standard OpenGL dovoluje výrobcům implementovat na jejich zařízení další funkce korespondující k novým technologiím a umožnit k nim přístup skrze rozšíření. Rozšíření je potom dodáváno ve dvou částech. První je hlavičkový soubor, který obsahuje prototypy funkcí extenze, a druhý je ovladač zařízení. Více výrobců může přistoupit k implementaci stejné funkce, tato může být poté „posvěcena“ ARBem a stane se z ní součást nového standardu. První takovou extenzí byla GL_ARB_multitexture, která se z pozice nepovinného rozšíření stala standardem počínaje OpenGL API 1.4.
Nad OpenGL je postaveno několik knihoven, které ho doplňují:

OpenGL Utility Library (GLU) - umožňuje využívat tesselátory (rozložení nekonvexních polygonů na trojúhelníky), evaluátory (výpočet souřadnic bodů ležících na parametrických plochách) a vykreslovat kvadriky (koule, válce, kužely a disky) Nastavuje matice pro speciální pohledové a projekční orientace. Příkazy knihovny GLU začínají prefixem *glu*.

The OpenGL Utility Toolkit (GLUT) - tvoří doplněk ke grafické knihovně OpenGL. Základem této nadstavbové knihovny je podpora pro práci s okny (včetně zpracování událostí), vyskakovacími menu a písmem. Tyto činnosti totiž nejsou v knihovně OpenGL přímo podporovány - důvodem je snaha o co největší zachování platformové nezávislosti. Funkce pro práci s okny či menu, které jsou systémově závislé, se dříve, tj. v době, kdy knihovna GLUT neexistovala, musely naprogramovat pro každý operační systém (resp. jeho grafickou nadstavbu) zvlášť, což od vývojáře aplikace vyžadovalo podrobnou znalost funkcí daného operačního systému, grafické nadstavby a správce oken. Příkazy knihovny GLUT začínají prefixem *glut*.

OpenGL User Interface Library (GLUI) je C++ knihovna uživatelského rozhraní založená na OpenGL Utility Toolkitu (GLUT), která přináší do OpenGL aplikací prvky jako tlačítka, checkboxy, radio buttony, apod. Je systémově nezávislá, spoléhá ve všech systémových závislostech (jako je obsluha okna nebo periférií) na GLUT.

Pokud se toužíte dozvědět o OpenGL více, doporučil bych článek Daniela Čecha OpenGL – referát na praktikum z informatiky, který naleznete na adrese http://nehe.ceske-hry.cz/cl_gl_referat.pdf.

4.2 Herní problematika

Vytvoření počítačové hry je velmi komplexní záležitost. Implementace musí zahrnovat mnohem více než jen problematiku počítačové grafiky. Základní věci, které obvykle hra postihuje, jsou následující:
Audio-vizuální stránka – Grafický výstup na monitoru, či jiném zobrazovacím zařízení, je nutnost. Grafika je první, co hráče zaujme, neměla by se však stát tím nejdůležitějším. Hra by měla mít především dobrou hratelnost a být zábavná. V našem případě nám ale jde hlavně o demonstrativní stránku věci, takže na implementaci grafiky klademe velký důraz. Nedílnou součástí herní aplikace je také ozvučení.
Fyzikální model – Pokud hra nespadá do kategorie logický hříček či tahových her a naopak se snaží o zobrazení reálného nebo vysněného světa, je nutné, aby v tomto světě nějaký způsobem pracovaly fyzikální zákony. Nemusíme postihnout veškeré působení fyziky, to by bylo nesmírně časově náročné a ani by to nebylo možné, je však nutné pokrýt tu část fyziky, která je podstatná pro simulace v aplikaci. Přesnost a věrnost výpočtů pak volíme podle požadované přesnosti simulace a podle

požadovaného výkonu aplikace. Do kategorie fyzikálních výpočtů lze začlenit i detekci kolizí a reakce na ně.

Herní logika – Zábavnost hry spočívá ve snažení se o dosažení nějakého cíle. Určení cíle a postup, jak ho dosáhnout, definují pravidla hry, o jejich dodržování a provádění se pak stará herní logika. Algoritmy herní logiky tvoří hru jako takovou.

Umělá inteligence – Soupeření bývá častým klíčem k zábavě, jak nám dokazují mnohá sportovní odvětví. Ne vždy je však lidský soupeř k dispozici a na jeho místo musí nastoupit počítač. Algoritmy umělé inteligence musí být schopné dovést počítačem reprezentovaného hráče k cíli skrze nástrahy herní logiku. Algoritmy by měli vykazovat jistou dávku náhodnosti a nepředvídatelnosti a hlavně musí dát lidskému hráči možnost vyhrát. Nejjednodušším způsobem implementace umělé inteligence do závodní hry, je dát počítačem řízenému vozidlu systém bodů, po kterých se má pohybovat, a jakou rychlostí, se má pohybovat. Pokud takovému algoritmu nastavíme ideální hodnoty, lidský hráč by v takovém případě mohl dosáhnout cíle maximálně současně s počítačovým protivníkem nebo pomaleji. A to je poněkud demotivující.

4.3 Načítání externích dat

Načítání externích dat přináší mnoho výhod jako je například možnost změny dat bez nutnosti rekompilace programu. Největší výhoda však spočívá v možnosti využití výkonných grafických nebo zvukových aplikací pro jejich editaci. Existují spousty formátů od těch nejjednodušších (.RAW), kdy stačí soubor nahrát do paměti tak jak je, přes mírně složitější (.TGA, .BMP), které se ještě dají nahrát vlastními silami, až po velmi složité formáty (.JPG apod.), kdy je rozumné rovnou využít cizí knihovnu. To samé platí i pro nejrůznější 3D modely z CAD/CAM programů, 3D Studia MAX, Milkshape 3D a z dalšího modelovacího softwaru. Orientovat se v načítání externích dat bývá opravdu složité, zvlášť v případech, kdy ani není znám přesný formát souboru, protože nebyl danou firmou vůbec neuvolněn. Před samotným psaním kódu je nezbytně nutné provést důkladnou analýzu daného formátu.

4.3.1 Načítání .3DS modelů

3ds soubor [11] obsahuje množství informací použitých k popsání každého detailu 3D scény složené z jednoho nebo více objektů. 3ds soubor obsahuje množství bloků, kterým říkáme „chunky“. Slovo chunk lze přeložit jako dávka informace. Tyto dávky informací 3ds souboru obsahují vše nezbytné k popsání scény: jméno každého objektu, koordináty vertexů, mapové koordináty textur, seznam polygonů, informace o barvách, klíčové snímky animací atd.

Chunky nemají lineární strukturu, to znamená, že některé mohou být závislé na jiných a mohou být přečteny pouze, pokud byl napřed přečten rodičovský chunk. Není nutné přečíst všechny chunky, stačí si vybrat ty, které nás zajímají.

Chunk je složen ze 4 polí:

Identifikátor – hexadecimální 2-bytové číslo identifikující chunk. Podle této informace hned poznáme, jestli je pro nás chunk důležitý. Jestliže ano, vytáhneme z něho pro nás účelné informace a pokud je to potřeba, tak i z jeho potomků. Jestliže chunk nepotřebujeme, jednoduše ho přeskočíme za použití informace o délce chunku.

Délka chunku – 4-bytové číslo, které je součtem velikosti chunku a všech jeho potomků.

Data chunku - pole proměnné délky obsahující data modelu.

Sub-chunky - potomci chunku

Přehled offsetů a délek chunků je shrnut v následující tabulce:

Offset	Délka	Obsah pole
0	2	Identifikátor chunku
2	4	Délka chunku: data chunku + sub-chunky(6+n+m)
6	n	Data
6+n	m	Sub-chunky

Tabulka 1: Offsety a délky chunků

Na posledním řádku je vidět, jak jsou chunky na sobě závislé: každý potomek může obsahovat pole potomků rodiče.

V tabulce 2 je uvedeno několik základních chunků, jejich hierarchie, identifikátory a umístění v souboru. 3ds soubor obsahuje velké množství takovýchto chunků, velká část z nich není zdokumentována.

Struktury 3ds loaderu:

Strukturu vertexu tvoří 3 floatové hodnoty

```
typedef struct{
    float x,y,z;
}vertex_type;
```

Strukturu polygonu tvoří 3 unsigned short (můžeme uložit jako integer)

```
typedef struct{
    int a,b,c;
}polygon_type;
```

Strukturu UV mapových koordinátů tvoří 2 floaty

```
typedef struct{
    float u,v;
}mapcoord_type;
```

Struktura 3ds objektu v programu

```
typedef struct {

    char name[20];       // jméno objektu
    int vertices_qty;    // počet vertexů objektu
    int polygons_qty;    // počet polygonů

    vertex_type vertex[MAX_VERTICES];   // pole vertexů
    vertex_type normal[MAX_VERTICES];   // pole normál vertexů
    polygon_type polygon[MAX_POLYGONS]; // pole polygonů
    mapcoord_type mapcoord[MAX_VERTICES]; // pole UV koordinátů textur
    int id_texture;       // identifikátor textury

} obj_type, *obj_type_ptr;
```

Přehled základních chunků:

MAIN CHUNK	
Identifikátor	0x4d4d
Délka	0 + délka sub-chunků
Nadřazený chunk	Žádný
Sub-chunky	3D EDITOR CHUNK
Data	Žádná

3D EDITOR CHUNK	
Identifikátor	0x3D3D
Délka	0 + délka sub-chunků
Nadřazený chunk	MAIN CHUNK
Sub-chunky	OBJECT BLOCK, MATERIAL BLOCK, KEYFRAMER CHUNK
Data	Žádná

OBJECT BLOCK	
Identifikátor	0x4000
Délka	Velikost jména + délka sub-chunků
Nadřazený chunk	3D EDITOR CHUNK
Sub-chunky	TRIANGULAR MESH, LIGHT, CAMERA
Data	Jméno objektu

TRIANGULAR MESH	
Identifikátor	0x4100
Délka	0 + délka sub-chunků
Nadřazený chunk	OBJECT BLOCK
Sub-chunky	VERTICES LIST, FACES DESCRIPTION, MAPPING COORDINATES LIST
Data	Žádná

VERTICES LIST	
Identifikátor	0x4110
Délka	proměnná + délka sub-chunků
Nadřazený chunk	TRIANGULAR MESH
Sub-chunky	Žádná
Data	Vertices number (unsigned short)
	Vertices list: x1,y1,z1,x2,y2,z2 etc. (pro každý vertex: 3*float)
FACES DESCRIPTION	
Identifikátor	0x4120
Délka	proměnná + délka sub-chunků
Nadřazený chunk	TRIANGULAR MESH
Sub-chunky	FACES MATERIAL
Data	Polygons number (unsigned short)
	Polygons list: a1,b1,c1,a2,b2,c2 atd. (pro každý bod: 3*unsigned short)
	Face flag: face options, sides visibility atd. (unsigned short)
MAPPING COORDINATES LIST	
Identifikátor	0x4140
Délka	proměnná + délka sub-chunků
Nadřazený chunk	TRIANGULAR MESH
Sub-chunky	SMOOTHING GROUP LIST
Data	**Vertices number (unsigned short)**
	Mapping coordinates list: u1,v1,u2,v2 atc. (pro každý vertex: 2*float)

Tabulka 2: Přehled základních chunků

Přestože z Main chunku, 3D editor chunku a Triangularmeshe nečteme žádná data, je potřeba tyto chunky přečíst, protože jsou nadřazenými chunky pro chunky se seznamem vertexů, polygonů a s UV koordináty, a jak již bylo řečeno, potomci jsou závislí na rodičovských chuncích.

4.3.2 Textury

Aplikace bude pracovat texturami formátu *.BMP a *.TGA. Tyto formáty byly vybrány pro svou jednoduchost. Jejich načtení vyžaduje rozparsování hlavičky, data jsou pak posloupnost následujících bytů.

BMP (Microsoft Windows Bitmap) [12] je počítačový formát pro ukládání rastrové grafiky. Výhodou tohoto formátu je jeho extrémní jednoduchost a dobrá dokumentovanost, a že jeho volné použití není znemožněno patentovou ochranou. Díky tomu jej dokáže snadno číst i zapisovat drtivá většina grafických editorů v mnoha různých operačních systémech. Obrázky BMP jsou ukládány

po jednotlivých pixelech, podle toho, kolik bitů je použito pro reprezentaci každého pixelu je možno rozlišit různé množství barev (tzv. barevná hloubka): 2 barvy (1 bit na pixel), 16 (4 bity), 256 (8 bitů), 65 536 (16 bitů), nebo 16,7 miliónů barev (24 bitů). Osmibitové obrázky mohou místo barev používat šedou škálu (256 odstínů šedi).

Soubory ve formátu BMP většinou nepoužívají žádnou kompresi (přestože existují i varianty používající kompresi RLE). Z tohoto důvodu jsou obvykle BMP soubory mnohem větší než obrázky stejného rozměru uložené ve formátech, které kompresi používají. Velikost nekomprimovaného obrázku v bajtech lze přibližně vypočítat podle vzorce:
*(šířka v pixelech) * (výška v pixelech) * (bitů na pixel / 8)*
K velikosti obrázku je třeba ještě připočítat velikost hlavičky souboru, která se liší dle jeho verze i dle použité barevné hloubky.

TGA (také označovaný jako **Targa**) [13] je další z řady ze souborových formátů pro ukládání rastrové počítačové grafiky. Formát vytvořila společnost Truevision, která se specializovala na výrobu obrazových adaptérů, tzv. videograbberů. Grabberů typu *Targa* existovalo několik typů a pro každý typ byla vytvořena varianta vlastního grafického formátu TGA (lišily se především v počtu bitů na pixel). Informace v souborech typu TGA jsou rozděleny do sekcí, přičemž pouze první je povinná – jde o informační hlavičku, jejíž velikost je vždy rovna 18 bytům. Jsou zde umístěny základní informace o obraze, zejména jeho rozlišení, způsob kódování barev pixelů a orientace obrázku. Data tohoto formátu mohou být kódována algoritmem RLE. 32-bytová verze TGA obsahuje informaci o alfa kanálu, čehož se v aplikaci bude využívat k blendingu. Alfa kanál bude například aplikován na textury stromů, skel atd.

RLE (run length encoding) [14] je bezztrátová komprese, která kóduje vstupní data tak, že kóduje posloupnosti stejných hodnot do dvojic (délka posloupnosti, hodnota). Účinnost komprese je silně závislá na charakteru vstupních dat, která musí obsahovat delší sekvence stejných znaků, jinak výrazně účinnost komprese klesá. Při komprimaci nevhodného obrazu může velikost proudu z výstupu kóderu RLE být až dvojnásobná oproti vstupu. Naproti tomu výhodou je, pokud obraz obsahuje velké plochy stejné barvy. Například 32-bytová textura modelu auta o rozměrech 512 x 512 pixelů bez komprese zabírá přesně jeden megabyte, po kompresi je to pouze 367 kB.

4.4 Audio-vizuální zpracování

4.4.1 Skybox

Model je potřeba umístit do nějaké scény. Jestliže se jedná o venkovní scénu, bývá obalena tzv. skyboxem. Skybox je v podstatě kostka, na jejíchž stěnách jsou naneseny textury zobrazující okolí. Někdy se používají i jiné tvary než krychle, často kvádry nebo různé kulovité kvadriky (pro ty se používá výraz **skydome**).

4.4.2 Kamera

Ve hře bude možno přepínat mezi několika kamerami. Všechny kamery, kromě kamery používané pro vývoj (je ovládána myší), budou pracovat za pomoci funkce *gluLookAt()*.

Specifikace funkce *gluLookAt():*

void ***gluLookAt****(GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ,*

```
GLdouble centerX, GLdouble centerY, GLdouble centerZ,
GLdouble upX, GLdouble upY, GLdouble upZ )
```

4.4.3 Animace

Pokud chceme hráče vtáhnout do děje, je vhodné atmosféru hry nebo její děj nastínit animací. Existují v podstatě 2 možnosti, jak animaci přehrát. Zaprvé můžeme vykreslovat požadovanou scénu za pomoci našeho enginu. Jestliže nám kvalita zobrazení nestačí, máme tu druhou možnost - přehrání filmové animace z video souboru.

Aplikace bude přehrávat video soubory ve formátu *.avi. Pro usnadnění práce s formátem využijeme externí knihovny pro práci s videem z dílny firmy Microsoft *vfw.h* - video for Windows.

4.4.4 Framebuffer

Framebuffer [15] je část paměti, do které OpenGL vykresluje výsledný obrázek. Skládá se z několika dílčích bufferů, z nichž každý je určen pro jednu nebo více specifických operací, které lze při vykreslování scény provádět. Do framebufferu se zapisují, nebo se z něho čtou, takzvané *fragmenty*, což jsou pixely, které kromě své barvy a průhlednosti obsahují i informace o hloubce (vzdálenosti od kamery), popř. i další informace. Na každý buffer se můžeme dívat jako na paměť, v níž jsou uložena rastrová data (pixely), která mají podle určení bufferu svůj specifický význam. Buffer lze vymazat nebo přečíst do hlavní paměti počítače, popř. do něj data přímo zapsat, což se používá například při programování některých grafických efektů.

Při práci s OpenGL můžeme přímo či nepřímo používat čtyři typy bufferů:

1. *Color buffer(s)* - barvový nebo více barvových bufferů
2. *Depth buffer* - paměť hloubky, nazývaný také Z-buffer
3. *Stencil buffer* - paměť šablony
4. *Accumulation buffer* - akumulační buffer

4.4.5 Color buffer(s)

V barvovém bufferu (nebo bufferech) je uložena barevná informace o vykreslované scéně. Jeden z barvových bufferů je vždy zobrazen na obrazovce. Barvový buffer je v nejjednodušším případě pouze jeden, avšak může jich být nainicializováno několik. Na naprosté většině grafických akcelerátorů lze vytvořit minimálně dva barevné buffery, které lze využít při animacích k takzvanému *double bufferingu.* V případě double bufferingu jsou tyto dva barevné buffery nazývány přední buffer (*front buffer*) a zadní buffer (*back buffer*). Princip spočívá v tom, že do jednoho bufferu se provádí vykreslování a druhý je zobrazen na obrazovce. Po dokončení vykreslování se funkce obou příkazem *glutSwapBuffers()* prohodí, tj. první bude zobrazen a do druhého se bude vykreslovat. Důležité je, že double buffering je většinou hardwarově podporován grafickou kartou, proto je přehození bufferů otázkou několika málo instrukcí a nemusí se provádět časově náročný přenos dat z druhého do prvního bufferu.

V barevném bufferu mohou být jednotlivé pixely uloženy buď ve formátu RGB (true-color), nebo může být použit index-color (paletový) režim, kdy pixely neobsahují přímo barvu, ale pouze index do barevné palety.

4.4.6 Depth buffer

Depth buffer (paměť hloubky) je někdy také nazýván Z-buffer. Jedná se o paměť, v níž jsou většinou uloženy informace zajišťující vykreslení pouze viditelných částí těles, tj. vzdálenější plošky jsou překryty ploškami bližšími. Funkci a význam paměti hloubky lze však při vykreslování průběžně měnit, takže je možné naprogramovat některé zajímavé efekty, například zobrazení stínů.

4.4.7 Stencil buffer

Stencil buffer (paměť šablony) je používán pro určení, do kterých míst na obrazovce je povoleno vykreslování. Jedno z možných použití stencil bufferu je při vykreslování 2D grafického uživatelského rozhraní současně s trojrozměrnou scénou. Dalším, pokročilejším použitím stencil bufferu je algoritmus pro vykreslování zrcadlících ploch nebo pro tvorbu stereo obrázků na monitoru s použitím stereo brýlí (dva pohledy na scénu snímané ze dvou bodů zobrazené prokládaně na jedné obrazovce).

4.4.8 Accumulation buffer

Accumulation buffer obsahuje, podobně jako color buffer, rastrový obrázek. Do akumulačního bufferu je však možno akumulovat (např. sečíst) více obrázků. Akumulační buffer se používá například při programování efektu rozmazání pohybem - takzvaný *motion blur*.

Funkce akumulačního bufferu je dostupná pouze při používání RGB (true-color) režimu. V index-color (paletových) režimech nelze akumulační buffer použít.

4.4.9 Částicové systémy

Částicové systémy se v počítačové grafice používají na programování mnoha efektů. Aplikace vytváří za pomoci částicových systémů efekt plamenů a efekt kouře. Částicové systémy je vhodné kombinovat s technikou billboardingu. Každá částice má stanovenou svoji délku života, rychlost vyhasínání částice, pozici emitoru, směr pohybu částice a způsob působení sil na částici. Struktura částice tedy vypadá následovně:

```
typedef struct {                    // Vytvoření struktury po částici
      float life;                   // Délka života částice
      float fade;                   // Rychlost vyhasínání
      float x;                      // Pozice emitoru v ose x
      float y;                      // Pozice emitoru v ose y
      float z;                      // Pozice emitoru v ose z
      float xi;                     // Směr pohybu částice v ose x
      float yi;                     // Směr pohybu částice v ose y
      float zi;                     // Směr pohybu částice v ose z
      float xg;                     // Působení síly v ose x
      float yg;                     // Působení síly v ose y
      float zg;                     // Působení síly v ose z
} particles;
```

4.4.10 Ozvučení

Ozvučení sice nespadá do oboru počítačové grafiky, ale neodmyslitelně patří ke scéně. Aplikace využívá k přehrávání hudby knihovnu FMOD.

FMOD je audio knihovna, která umožňuje přehrávat hudební soubory různých formátů na mnoha platformách. Je ve dvou verzích FMOD a FMOD Ex. „Ex“ je označení dané pro sérii 4.x této knihovny. FMOD samotná odkazuje na verzi 3.75, která postrádá mnoho z podporovaných formátů/platforem prezentovaných ve FMOD Ex, ale je stále využívána. Mnoho herních a softwarových vývojářů používají FMOD k poskytnutí audio funkcí jejich aplikacím.

4.5 Detekce kolizí

Detekce kolizí představuje zásadní problém v počítačových animacích, fyzikálních simulacích, geometrickém modelování a robotice. Detekce kolizí hledá stavy, kdy dva objekty v prostoru pronikají do sebe. Detekce je velmi náročná na strojový čas, proto je důležité zvolit jakou metodou a pro které objekty bude počítána. Při malé frekvenci počítání kolizí u rychle pohybujících objektů může nastat případ, kdy těleso může projít skrze jiné bez detekování kolize, proto je podstatné i jak často se kolize počítají. Výpočty kolizí jsou založené na výpočtech matematické analýzy [16].

4.5.1 Kolize bodu a koule

Asi nejjednodušší použitelnou kolizí v prostoru je kolize bodu a koule. Kolize nastává tehdy, jsou-li všechny souřadnice bodu a středu koule ve vzdálenosti rovné nebo menší poloměru koule. Vzdálenost bodu od středu koule můžeme jednoduše vypočítat užitím Pythagorovy věty rozšířené o třetí rozměr. Pokud platí $px^2 + py^2 + pz^2 \leq poloměr^2$, pak nastala kolize.

4.5.2 Kolize bodu a elipsoidu

U těles, jejichž tvar není ve všech třech osách stejný, musíme mít uloženou orientaci v prostoru v rotační matici tělesa. Kromě rotační matice bude struktura elipsoidu obsahovat informace o souřadnicích středu, poloměr elipsoidu a procentuální vyjádření poloměrů. Tzn. osa s největším poloměrem bude nabývat hodnoty 1.00. Princip zjišťování kolize bodu a elipsoidu je pak podobný s předchozím případem. Napřed určíme vektor v, směřující ze středu elipsoidu k testovanému bodu. Poté vektor vynásobíme transponovanou maticí pro otočení, čímž vyrušíme rotaci elipsoidu v prostoru. Nakonec překonvertujeme elipsoid na kouli o poloměru *r* a složky vektoru změníme v poměru jednotlivých poloměrů *r1, r2, r3*. Nyní už můžeme provést kolizní test bodu a koule.

4.5.3 Kolize bodu a kvádru

Výpočet kolize bodu a kvádru je postup podobný jako při výpočtu kolize bodu s elipsoidem. Určíme vektor v, směřující ze středu kvádru do testovaného bodu. Vektor vynásobíme transponovanou maticí s informacemi o otočení kvádru. Nyní zbývá jen otestovat, zda se bod nachází uvnitř kvádru. To je pravda, pokud platí: $|x| < \frac{šířka}{2} AND |y| < \frac{výška}{2} AND\ |z| < \frac{délka}{2}$.

5 Implementace

Kapitola popisuje některé detaily implementace technik uvedených v kapitole 4 – návrh a ukazuje dosažené výsledky. Popisuje řešení načítání dat, vizuálních zpracování, detekcí kolizí a především herní logiky, jejíž algoritmus je rozebrán podobněji.

5.1 Schéma řešení

Vzájemná interakce modulů programu je znázorněna schématem 5.1.

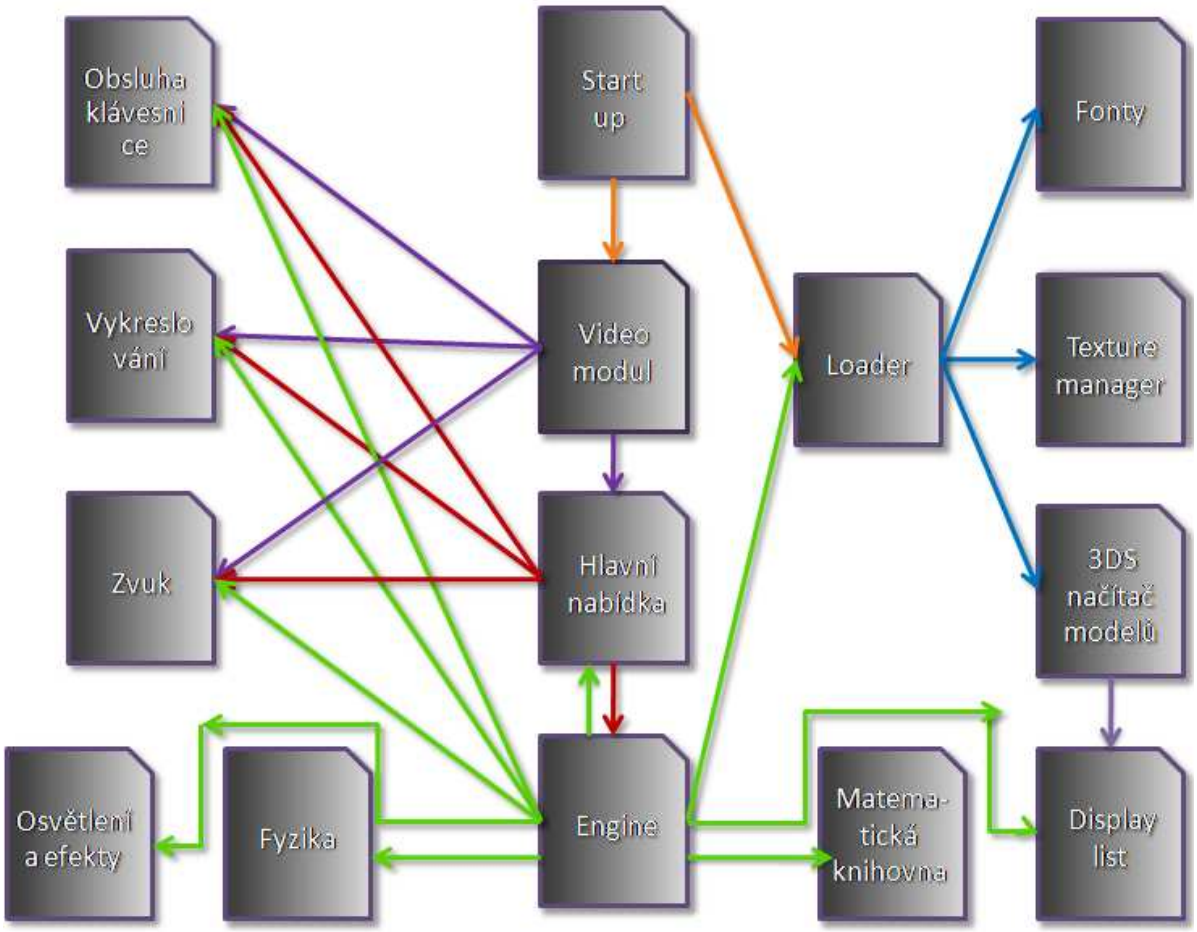


Schéma 5.1

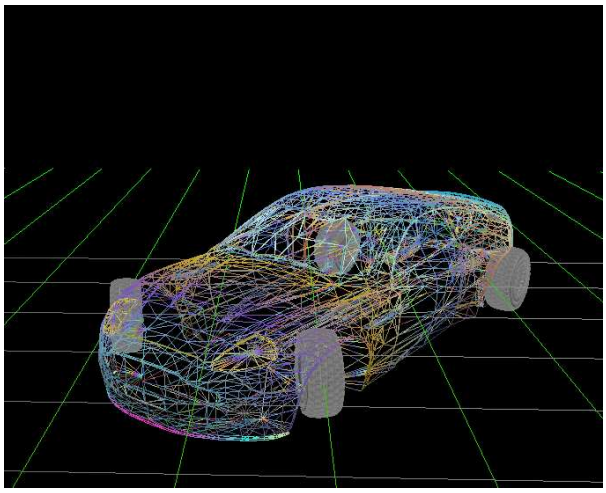
5.2 Načítání externích dat

Výsledná aplikace čte z 3ds souboru data o vertexech, polygonech a UV koordinátech textur. Do budoucna se nabízí mnoho rozšíření tohoto modulu v podobě načítání scény složené z více podobjektů, načítání textur, materiálů, či dokonce animací. Mapování textur je nyní prováděno za pomoci UV koordinátů, vlastnosti materiálu jsou nadefinové v display listu.

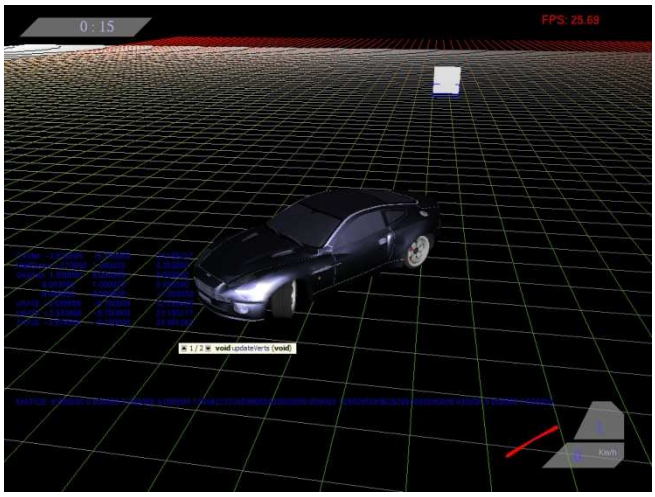
Při načítání modelů jsem narazil na problém různých poměrů velikostí objektů. Například kolo u auta může být dvakrát tak velké jak celé auto, dveře mohou být větší než dům. Tento problém jsem řešil floatovou proměnnou, která vstupuje do funkce jako parametr a dělí vzdálenost vertexu od středu v každé ose. Tím jsem docílil toho, že načtený model může být x-krát zmenšen nebo zvětšen.

Vykreslování modelů je prováděno skrz display listy. To sice zvýší výkon aplikace oproti běžnému vykreslování primitiv v *immediate* módu, ale zvýšení výkonu aplikace má značné rezervy. Proto jsem loader rozšířil i funkci, která uloží vertexy, normály a texturové koordináty do vertexových polí. Zatím jsou vertexová pole vykreslována také skrze display list, nedojde tudíž ke zvýšení výkonu, ale do budoucna je takto loader připraven pro práce s vertex bufferem grafické karty.

Vývoj 3D loaderu je dobře viditelný na vývoji modelu auta:



Obrázek 5.1: Loader čte vertexová data



Obrázek 5.2: Vypočet normál, Gouraudhovo stínování, materiály

Obrázek 5.2: kompletně otexturovaný model bez blendingu

Obrázek 5.4:kompletně otexturovaný model s blendingem

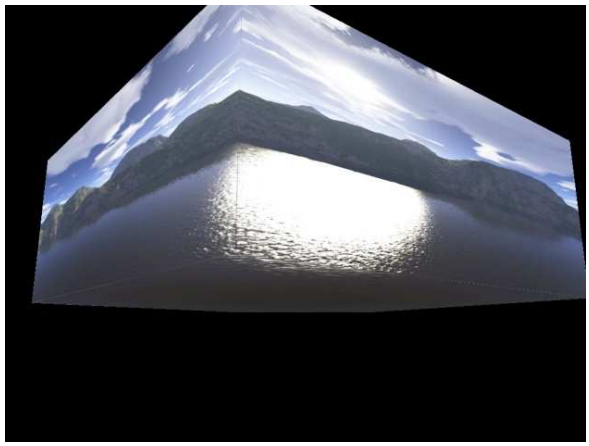
5.3 Audio-vizuální zpracování

5.3.1 Skybox

Implementace skyboxu je jednoduchá záležitost. Jediný nepříjemný jev, který se objevil, byly viditelné hrany krychle skyboxu. To jsem řešil tím, že jsem posunul, při zachování velikosti stran, každou stěnu o něco blíže ke středu. Texturu stěny by bylo dobré obehnat rámečkem, aby byl přechod mezi stěnami plynulý, ale při velké velikosti skyboxu to není potřeba. Dále je jsem nechal skybox pohybovat spolu s vozidlem a tak není možné skybox nikdy opustit.

Obrázek 5.5: Skybox

Obrázek 5.6: Skybox po odstranění viditelných hran



Obrázek 5.7: Venkovní pohled na skybox

5.3.2 Kamera

Včetně vývojářské kamery ovládané myší hra obsahuje 5 pohledů kamer. Implicitní kamera sleduje auto z dálky, pokud zatáčí, plynule se natočí až do určitého úhlu, poté se stejným způsobem vrací do výchozí pozice. Do budoucna se mi jeví výhodně implementovat kameru pracující na stejném principu jako například stopy po pneumatikách (kamera držící přesnou stopu auta). Pohled kamery by tak dynamicky reagoval na rychlost vozidla, při vyšších rychlostech by došlo k oddálení, při nižších k přiblížení kamery, a dobře tak uvozoval pocit sledování pohybu.

5.3.3 Animace

Časovač hry pracuje s dvěma proměnnými. Jedna je časovačem inkrementována po celý běh herní fáze, druhá pouze v případě, že hra není pozastavena. Hlavním účelem této proměnné je měřit čas závodu, ale dá se jí i dobře využít pro in-game animace. Proměnná game_time poté animaci i řídí. Zde je příklad práce s kamerou při in-game animaci:

```
gluLookAt(7*sin(RAD(game_time*0.4))+car->position_wc.x, -0.2,
          50*cos(RAD(game_time*0.4)) -car->position_wc.z,
          car->position_wc.x, 0.0, -position_wc.z,
          0.25, 0.9, 0.15);
```

Stejným způsobem lze ovládat i jiné prvky ve scéně. Užitím goniometrických funkcí je dosaženo plynulého pohybu kamery po oblouku. Změnou up vektoru získáme líbivé úhly pohledu kamery.

Obrázek 5.8: Ukázka in-game animace

Aplikace přehrává video soubory ve formátu *.avi. Přehrávač videa využívá pomoci knihovny pro práci s videem z dílny firmy Microsoft *vfw.h* - video for Windows. Více informací o implementaci se nachází v tutoriálu.

Obrázek 5.9: Ukázka video animace z úvodního intra

5.3.4 Využití bufferů

Aplikace využívá všech základních bufferů. Odraz vozidla na mokré vozovce byl vytvořen za použití stencil buffer:

Obrázek 5.10: Využití stencil bufferu k zobrazení odrazu šablony

Za pomoci akumulační bufferu byl vytvořen motion blur – efekt rozmazání pohybem:

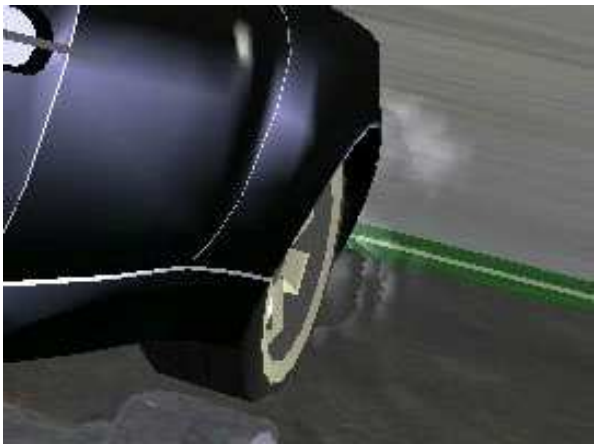
Obrázek 5.11: motion blur - statická kamera

Obrázek 5.15.3: motion blur - kamera pohybující se spolu s vozidlem

5.3.5 Částicové systémy

Implementovány byly dva částicové systémy – efekt plamene při nitro boostu a kouř od pneumatik. Rychlost zpomalení částic je určena konkrétním efektem. Viditelnost (alfa kanál) částic je ovlivněna jejich životním stádiem. Na částice kouře je nanesena textura využívající blending. Částice plamene texturou pokryty nejsou, jejich barva je generována (odstíny od žluté až po oranžovou). Bylo by možné nastavovat barvu podle stádia života částice a podle vzdálenosti od středu emitoru, tím se dá například docílit, že plamen bude mít v místě vzniku žlutou barvu a postupně bude přecházet do oranžových odstínu. Toto lze řešit i vhodnou texturou.

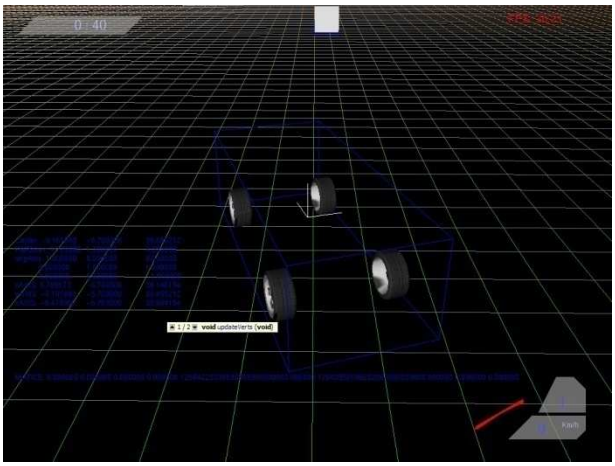
Obrázek 5.13: částicové systémy - plamen



Obrázek 5.14: částicové systémy - dým

5.4 Detekce kolizí

Detekce kolicí je řešena za pomoci algoritmu pro detekování kolize bodu a kvádru. Vozidlo je obaleno boxem jehož vrcholy jsou otáčeny spolu se souřadným systémem vozidla. Pro každý vrchol se pak provádí kolizní test s prostorově orientovanými kolizními boxy obalujícími objekty. Jedná se o zjednodušenou verzi testování kolizí mezi dvěma kolizními boxy.



Obrázek 5.15: Kolizní box obalující vozidlo

Obrázek 5.16: Objekty dráhy obalené kolizními boxy

5.5 Herní logika

Aby se hra dala hrát, musíme jí dát do vínku nějaký smysl. Tento smysl můžeme pojmout jako úkol dostat hru z počátečního stavu A do koncového stavu B. Herní logiku si pak můžeme představit jako stavový automat, který řídí přechod mezi stavy podle určitých pravidel. Jako příklad můžeme použít všem dobře známou hru šachy. Jako počáteční stav si můžeme představit začátek hry, kdy jsou všechny figurky rozmístěny na polích šachovnice ve svých výchozích pozicích. Konečným stavem hry je pak stav, kdy se jeden hráč přemístí jakoukoliv svoji figurku na pole, na kterém se nachází figurka krále druhého hráče. Hra může nabývat mnoha různých stavů, přechod mezi stavy (pohyb figurek) se však řídí jasnými pravidly.

U závodních her bývá obvykle počátečním stavem vozidlo na startovní čáře v čase nula a koncovým stavem je dosažení cíle. K výhře pak může být zapotřebí dosáhnout koncového stavu do určitého času nebo ho dosáhnout dříve než soupeři. Mezistavy pak mohou být například kontrolní úseky na dráze nebo například dokončení jednoho z několika kol.

Herní logika se stejně jako například fyzikální simulace počítá ve smyčce nezávisle na vykreslování. Samozřejmě je možné nastavit vlastní časovač pro fyziku, pro kolize, pro herní logiku, nebo je sdružit do jedné smyčky. Časovač ve hře vypadá zjednodušeně takto:

```
void časovačHry (int čas) {
    časStavu++;

    if(!pauza) {
        časHry++;
        if(spočtiKolize(všechnyKolizníObjekty) != 0)
            proveďReakceNaKolize(kolidujícíObjekty);
        spočtiFyziku(auto, delta_t);
        proveďHerníLogiku(auto, trať);
    }

    if(stavHry == HRA_BĚŽÍ)
        glutTimerFunc(25, časovačHry, 0);
}
```

Po načtení hry se začne inkrementovat proměnná *state_time*. Pokud není hra pozastavena stisknutím klávesy escape, která vyvolá herní menu, inkrementuje se i, pro herní logiku důležitá, proměnná *game_time*. Ta řídí samotný průběh hry. Zpočátku řídí kameru a in-game animaci, kdy se auto přibližuje ke startovní čáře. Následuje animace odpočtu (3, 2, 1, go). Na odpočet je možné rovnou přeskočit stisknutím klávesy enter – *game_time* se navýší na hodnotu, kterou by proměnná měla po skončení in-game animace. Po skončení odpočtu se začne za pomoci této proměnné počítat čas závodu a odemknou se herní funkce jako počítání fyzikální simulace (herní klávesy jsou odemčeny již při odpočtu). Závod začíná.

Každá trať (původně dvě, ve finální verzi byla ponechána pouze jediná) je složena z několika checkpointů, kontrolních bodů, kterými musí závodní auto projet, aby mu bylo kolo uznáno. Checkpointy nejsou body v pravém slova smyslu, jedná se vlastně o boxy, jež se vkládají do pole funkcí *void setCheckpoint(track * Track, float x, float y, float z, float w, float h, float l, float angle).* Startovací a zároveň cílový checkpoint je v poli checkpointů první a při vykreslení toho pole má žlutou barvu (ostatní jsou znázorněny zeleně). Díky těmto „bodům“ není možné, aby se hráč do cíle vrátil bez projetí okruhu a bylo mu uznáno kolo. Checkpointy jsou rovnoměrně rozmístěny po celé trati, při jejich průjezdu je hráči ukázán aktuální čas kola nebo varovné hlášení, pokud projíždí nesprávným checkpointem.

Algoritmus pracuje následovně. Časovač volá funkci *void checkCheckpoints(track * Track, CAR * car),* která zkontroluje, zda se auto právě nenachází v jednom z checkpointů. Pro zjednodušení výpočtů, kontrolujeme vždy pouze přítomnost středu auta, tím stačí testovat pouze jeden bod. Pokud předpokládáme, že se kontrolní stanoviště nepřekrývají, mohli bychom algoritmus zefektivnit přerušením výpočtu v případě, že se auto již v nějaké kontrolní zóně nalézá. U každého checkpointu nás zajímá, jestli do něho auto právě vjelo, nebo jestli se v něm už nějakou dobu nalézá. Kontrolujeme tedy, zda je číslo posledního checkpointu o jedna nižší než číslo checkpointu, ve kterém se auto nalézá. V podstatě mohou nastat tyto případy: Zaprvé, číslo minulého checkpoitnu je o jedna nižší než číslo checkpointu, ve kterém se auto nalézá. Tzn. auto právě vjíždí do dalšího checkpointu na dráze. Provedeme tedy nutné procedury, jako přepsání hodnoty posledního checkpointu a nastavení času potřebného na jeho dosažení. Druhý případ, který může nastat, je, že číslo minulého checkpointu je rovno checkpointu, ve kterém se auto nalézá. To může znamenat dvojí, buď auto projíždí nově dosaženým checkpointem, nebo se vrátilo k poslednímu checkpointu, ale nedosáhlo ještě checkpointu nového. V takovém případě zobrazíme na obrazovce čas potřebný k dosažení tohoto kontrolního bodu, aby hráč věděl, jak dobře si počíná. Pokud nenastal ani jeden ze zmiňovaných případů, znamená to, že se auto pohybuje špatnou cestou (například se snaží projet okruh opačným směrem). V takovém případě na to hráče upozorní varovné hlášení. Tato procedura platí pro běžný kontrolní bod na trati. Trochu jinak vypadá procedura i startovního, resp. cílového checkpointu. Zde při správném vstupu (auto dokončilo kolo) ještě nastavíme čas kola a pokud je tento čas lepší než čas kola minulého, nastavíme nový nejlepší čas kola. Dále samozřejmě inkrementujeme počet kol. Pokud bylo zajeto poslední kolo, závod končí a hra pokračuje do další herní fáze. Jestliže počet kol ještě nedosáhl cílové hodnoty, zobrazí se při průjezdu startovní zónou hráči čas právě zajetého kola.

```
void checkCheckpoints(track * Track, CAR * car) {

   TrackHUD = 0;

   for(int i=0; i<check_num; i++) {

      if(testCollision(car->position_wc.x, 0.0, car->position_wc.y, &Track-
>checkpoints[i])) {

         if(i == 0) {

            if(Track->last_checkpoint == check_num - 1) {

               Track->last_lap = (game_time-380) - Track->current_time;
               Track->lap++;
               Track->current_time = (game_time-380);
               if(Track->best_lap > Track->last_lap || Track->best_lap == 0)
                  Track->best_lap = Track->last_lap;
               Track->last_checkpoint = i;
            }
            if(Track->last_checkpoint == 0 && Track->lap != 0) {
               if(Track->lap == LAP_NUM) {

                  loadEndOfRace();
                }
                else {
                   TrackHUD = 1;
                }
            }
            else if(Track->lap != 0) {
               TrackHUD = 3;
            }
         }
         else {

            if(Track->last_checkpoint == i - 1) {
               Track->last_checkpoint = i;
               Track->checkpoint_time = (game_time-380) – Track->current_time;
            }
            else if(Track->last_checkpoint == i) {
               TrackHUD = 2;
            }
            else {
               TrackHUD = 3;
            }
         }

      }
   }
}
```

TrackHUD1:
sprintf(lap_time_string,"LAP TIME: %d : %d", Track.last_lap/2400, (Track.last_lap/40)%60);
TrackHUD2:
sprintf(lap_time_string,"%d : %d", Track.checkpoint_time/2400, (Track.checkpoint_time/40)%60);
TrackHUD3:
renderBitmapString(420, 300, (void *)GLUT_BITMAP_TIMES_ROMAN_24, "WRONG CHECKPOINT");

5.6 Systém stavů

Hra, která hned po spuštění, umístí hráče na začátek mise a po jejím skončení se program ukončí, by dnešního náročného hráče asi příliš nezaujala. Proto je potřeba vytvořit systém stavů. Množina mnou použitých stavů vypadá takto:

GAME_BEGIN – začátek hry, zobrazení startup okna

GAME_MAIN_MENU – hlavní menu hry

GAME_RUNNING – hra samotná, závod

GAME_INTRO – počáteční filmové intro

GAME_LOADING –obrazovka načítání

RACE_OVER – stav po dokončení závodu

Přechod mezi stavy je prováděn skrze nahrávací funkce. Tyto funkce většinou mají stejný průběh. Uvolní paměť předešlého stavu, nahrají data stavu nového (textury, modely, zvuky), inicializují proměnné, provedou ostatní potřebné operace, změní hodnostu stavu na nový stav a spustí jeho timer. Timer původního stavu zaniká.

Vykreslovací smyčka vykresluje podle aktuálního stavu:

```
void RDraw(void) {
     if(GAME_STATE == GAME_BEGIN) {

        DrawStartup();
      }
      if(GAME_STATE == GAME_INTRO) {

          DrawIntro();
      }
      if(GAME_STATE == GAME_LOADING) {

          DrawLoading();
      }
     if(GAME_STATE == GAME_MAIN_MENU) {
            ⋮
```

6 Závěr

Cílem této práce bylo dát čtenáři základní povědomí o tvorbě počítačové grafiky a poskytnout mu v přiloženém tutoriálu návod, jak naimplementovat techniky, jakými jsou například načítání modelů, mapování textur, vytváření částicových systémů, práce s kamerou, přehrávání videa, využití grafických bufferů pro tvorbu různých efektů a mnoho dalších. Výsledná aplikace pak poslouží především jako ukázka toho, jak vše skloubit dohromady. To vše je dosaženo za pomoci nízko-úrovňových prostředků a umožňuje tak zájemci nahlédnout hlouběji do problematiky.

I přes časovou náročnost takového projektu, jakým je tvorba hry, se mi podařilo vytvořit aplikaci s relativně dobrou úrovní audiovizuálního zpracování. Dále se aplikace může pochlubit dobrou hratelností při zachování věrného fyzikálního modelu. Dokončit se podařilo i implementaci herní logiky, dá se tedy říci, že aplikace splňuje požadavky hry.

6.1 Rozšíření stávající práce

Práce na hře může být prakticky nekonečným úkolem. Stále je možné hru rozšiřovat, nejen o grafické efekty, ale i o level-design, příběh atd. S novými technologiemi přichází také nové možnosti, ale to již spadá spíše do řad profesionálních tvůrců her.

Osobně bych chtěl aplikaci rozšířit o mnoho vizuálních efektů, které jsem měl možnost nastudovat, ale nezbyl již čas na jejich implementaci. Jedná se například implementaci stencil shadow a soft shadow, porovnat rozdíly ve výsledném zobrazení a výkonu. Implementovány byly i techniky, které zatím nenašly své uplatnění. Například multitexturingu bych chtěl použít pro bump-mapping a reflection mapping, a značně tak vylepšit grafický vzhled aplikace. Využít implementovaný billboarding k tzv. „lens flare“ efektu. Rozšíření se nabízí skutečně mnoho a mnoho.

Pokud opomeneme grafickou stránku věci, chtěl bych v budoucnu hru obohatit i počítačem ovládaného soupeře. Nezbytným rozšířením se jeví také práce se soubory z důvodů ukládání konfigurací a pozic. Co se fyziky týče, chtěl bych rozšířit pohyb modelu o pohyb po výškové mapě a vylepšit systém kolizí.

Rozhodně by stálo za to, v případě projeveného zájmu, zveřejňovat nové verze hry spolu s tutoriálem pojednávajícím o nově implementovaných technikách na internet, a nabídnout tak zájemcům možnost rozšiřovat si své vzdělání v oblasti grafiky.

Literatura

[1] J. Žára, B. Beneš, J. Sochor, P. Felkel, *Moderní počítačová grafika*, Brno, Computer Press, 2004.

[2] Walnum, C. *Programujeme grafiku v Direct3D*. Brno, Computer Press, 2004.

[3] Lidický, B. *Několik poznámek k tvorbě počítačových her*, [on-line], [cit. 8.5.2008]. URL http://nehe.ceske-hry.cz/cl_sdl_hry.pdf

[4] Kršek, P. *Základy pocítacové grafiky IZG*, Studijní opora, [on-line], [cit. 8.5.2008].

[5] Vitulli, D. *Vectors, normals and OpenGL lighting*, [on-line], [cit. 8.5.2008]. URL http://www.spacesimulator.net/tut5_vectors_and_lighting.html

[6] Zajiček, O. *Stínování těles v počítačové grafice*, [on-line], [cit. 8.5.2008]. URL http://artax.karlin.mff.cuni.cz/~zajio1am/texts/practics/review7.html

[7] *OpenGL vedle DirectX stále žije*, [on-line], [cit. 8.5.2008]. URL http://www.ddworld.cz/linux/opengl-vedle-directx-stale-zije-vychazi-opengl-3-5.html

[8] Boháček, P. *Esej na téma „Co a k čemu je OpenGL*, [on-line], [cit. 8.5.2008]. URL http://www.bohacek.info/doc/2005-OGL.pdf

[9] Monster, M. *Car Physics for Games*, [CD-ROM] , 2003.

[10] Zuvich, T. *Vehicle Dynamics for Racing Games*, [CD-ROM]

[11] Vitulli, D. *3DS format file reader*, [on-line], [cit. 8.5.2008]. URL http://www.spacesimulator.net/tut4_3dsloader.html

[12] *Wikipedie: Otevřená encyklopedie*. [online], [cit. 8.5.2008]. URL http://cs.wikipedia.org/wiki/BMP

[13] *Wikipedie: Otevřená encyklopedie*. [online], [cit. 8.5.2008]. URL http://cs.wikipedia.org/wiki/TGA

[14] *Wikipedie: Otevřená encyklopedie*. [online], [cit. 8.5.2008]. URL http://cs.wikipedia.org/wiki/RLE

[15] Tišnovský, P. *Grafická knihovna OpenGL (12): vlastnosti framebufferu*. [online], [cit.8.5.2008]. URL http://www.root.cz/clanky/opengl-12-vlastnosti-framebufferu

[16] Pravda, J. *Jak vyzrát na kolize 2.díl - kolize v prostoru*. [online], [cit.8.5.2008]. URL http://www.builder.cz/art/cpp/kolize2.html

Seznam příloh

Příloha 1.: Ovládání programu

Úvodní intro

Enter – přeskočení intra, skok do hlavní nabídky

Hlavní nabídka

Šipky nahoru/dolů – vertikální pohyb v menu

Šipky doleva/doprava – horizontální pohyb v menu

Enter – potvrzení výběru

Hra

Šipka nahoru – akcelerace

Šipka dolů – brzda

Šipka doleva/doprava – natáčení kol

Mezerník – ruční brzda

n/N – nitro boost

c/C – změna kamery

Enter – přeskočení úvodní animace

Esc – vyvolání herní nabídky

Herní nabídka

Šipky nahoru/dolů – vertikální pohyb v menu

Enter – potvrzení výběru

Esc – návrat do hry

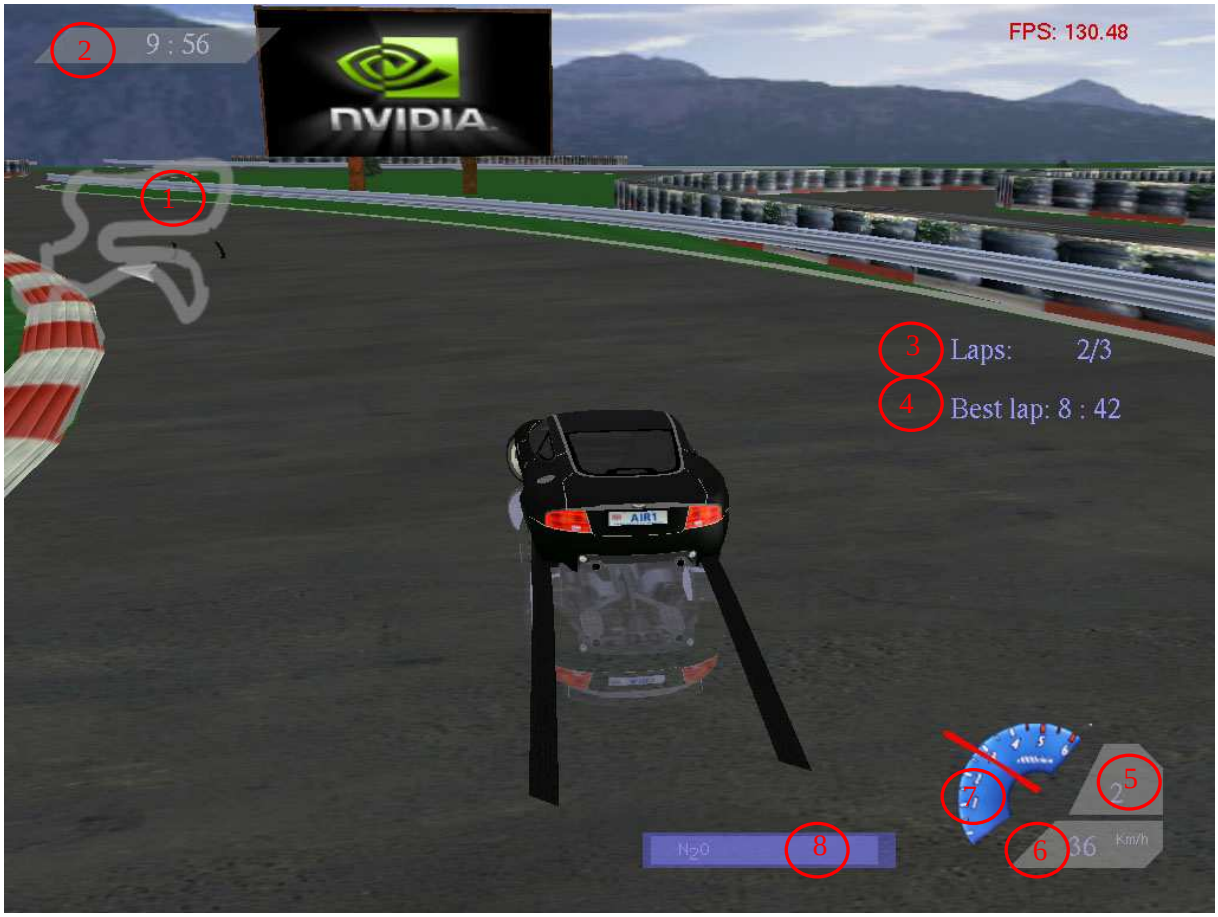
Konec závodu

Šipky nahoru/dolů – vertikální pohyb v menu

Enter – potvrzení výběru

Vývojářská kamera je ovládána myší, levé tlačítko slouží pro rotaci kamery, pravé tlačítko pro zoom.

Příloha 2.: HUD prvky



1 **–** minimapa (šipka ukazuje, kde se auto na trati právě nachází)

2 **–** celkový čas závodu

3 **–** aktuálně zajížděné kolo z celkového počtu kol

4 **–** čas nejlépe zajetého kola

5 **–** převodový stupeň

6 **–** aktuální rychlost

7 **-** otáčkoměr

8 **–** stav nádrže nitra

Příloha 3.: Schéma struktury 3DS souboru

```
MAIN3DS  (0x4D4D)
                 |
                 +--EDIT3DS  (0x3D3D)
                 |  |
                 |  +--EDIT_MATERIAL (0xAFFF)
                 |  |  |
                 |  |  +--MAT_NAME01 (0xA000) (See mli Doc)
                 |  |
                 |  +--EDIT_CONFIG1  (0x0100)
                 |  +--EDIT_CONFIG2  (0x3E3D)
                 |  +--EDIT_VIEW_P1  (0x7012)
                 |  |  |
                 |  |  +--TOP            (0x0001)
                 |  |  +--BOTTOM         (0x0002)
                 |  |  +--LEFT           (0x0003)
                 |  |  +--RIGHT          (0x0004)
                 |  |  +--FRONT          (0x0005)
                 |  |  +--BACK           (0x0006)
                 |  |  +--USER           (0x0007)
                 |  |  +--CAMERA         (0xFFFF)
                 |  |  +--LIGHT          (0x0009)
                 |  |  +--DISABLED       (0x0010)
                 |  |  +--BOGUS          (0x0011)
                 |  |
                 |  +--EDIT_VIEW_P2  (0x7011)
                 |  |  |
                 |  |  +--TOP            (0x0001)
                 |  |  +--BOTTOM         (0x0002)
                 |  |  +--LEFT           (0x0003)
                 |  |  +--RIGHT          (0x0004)
                 |  |  +--FRONT          (0x0005)
                 |  |  +--BACK           (0x0006)
                 |  |  +--USER           (0x0007)
                 |  |  +--CAMERA         (0xFFFF)
                 |  |  +--LIGHT          (0x0009)
                 |  |  +--DISABLED       (0x0010)
                 |  |  +--BOGUS          (0x0011)
                 |  |
                 |  +--EDIT_VIEW_P3  (0x7020)
                 |  +--EDIT_VIEW1    (0x7001)
                 |  +--EDIT_BACKGR   (0x1200)
                 |  +--EDIT_AMBIENT  (0x2100)
                 |  +--EDIT_OBJECT   (0x4000)
                 |  |  |
                 |  |  +--OBJ_TRIMESH   (0x4100)
                 |  |  |  |
                 |  |  |  +--TRI_VERTEXL          (0x4110)
                 |  |  |  +--TRI_VERTEXOPTIONS    (0x4111)
                 |  |  |  +--TRI_MAPPINGCOORS     (0x4140)
                 |  |  |  +--TRI_MAPPINGSTANDARD  (0x4170)
                 |  |  |  +--TRI_FACEL1           (0x4120)
                 |  |  |  |  |
                 |  |  |  |  +--TRI_SMOOTH            (0x4150)
                 |  |  |  |  +--TRI_MATERIAL          (0x4130)
                 |  |  |  |
                 |  |  |  +--TRI_LOCAL            (0x4160)
                 |  |  |  +--TRI_VISIBLE          (0x4165)
```

```
|   |   |
|   |   +--OBJ_LIGHT    (0x4600)
|   |   |   |
|   |   |   +--LIT_OFF              (0x4620)
|   |   |   +--LIT_SPOT             (0x4610)
|   |   |   +--LIT_UNKNWN01         (0x465A)
|   |   |
|   |   +--OBJ_CAMERA   (0x4700)
|   |   |   |
|   |   |   +--CAM_UNKNWN01         (0x4710)
|   |   |   +--CAM_UNKNWN02         (0x4720)
|   |   |
|   |   +--OBJ_UNKNWN01 (0x4710)
|   |   +--OBJ_UNKNWN02 (0x4720)
|   |
|   +--EDIT_UNKNW01  (0x1100)
|   +--EDIT_UNKNW02  (0x1201)
|   +--EDIT_UNKNW03  (0x1300)
|   +--EDIT_UNKNW04  (0x1400)
|   +--EDIT_UNKNW05  (0x1420)
|   +--EDIT_UNKNW06  (0x1450)
|   +--EDIT_UNKNW07  (0x1500)
|   +--EDIT_UNKNW08  (0x2200)
|   +--EDIT_UNKNW09  (0x2201)
|   +--EDIT_UNKNW10  (0x2210)
|   +--EDIT_UNKNW11  (0x2300)
|   +--EDIT_UNKNW12  (0x2302)
|   +--EDIT_UNKNW13  (0x2000)
|   +--EDIT_UNKNW14  (0xAFFF)
|
+--KEYF3DS (0xB000)
   |
   +--KEYF_UNKNWN01 (0xB00A)
   +--.............. (0x7001) ( viewport, same as editor )
   +--KEYF_FRAMES   (0xB008)
   +--KEYF_UNKNWN02 (0xB009)
   +--KEYF_OBJDES   (0xB002)
      |
      +--KEYF_OBJHIERARCH  (0xB010)
      +--KEYF_OBJDUMMYNAME (0xB011)
      +--KEYF_OBJUNKNWN01  (0xB013)
      +--KEYF_OBJUNKNWN02  (0xB014)
      +--KEYF_OBJUNKNWN03  (0xB015)
      +--KEYF_OBJPIVOT     (0xB020)
      +--KEYF_OBJUNKNWN04  (0xB021)
+--KEYF_OBJUNKNWN05  (0xB022)
```

Obrázek k příloze 3. : Většinu objektů okruhu tvoří importované 3DS modely